

The Design and Implementation of a Distributed File System based on Shared Network Storage

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Steven R. Soltis

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

August 1997

UMI Number: 9801013

UMI Microform 9801013
Copyright 1997, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

This is an authorized facsimile, made from the microfilm master copy of the original dissertation or master thesis published by UMI.

The bibliographic information for this thesis is contained in UMI's Dissertation Abstracts database, the only central source for accessing almost every doctoral dissertation accepted in North America since 1861.

UMI[®] Dissertation
Services

From:ProQuest
COMPANY

300 North Zeeb Road
P.O. Box 1346
Ann Arbor, Michigan 48106-1346 USA
800.521.0600 734.761.4700
web www.il.proquest.com

Printed in 2002 by digital xerographic process
on acid-free paper

DPGT

THIS PAGE BLANK (USPTO)

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

THIS PAGE BLANK (USPTO)

© Steven R. Soltis, 1997
All rights reserved.

UNIVERSITY OF MINNESOTA

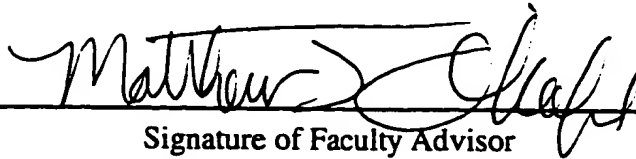
This is to certify that I have examined this bound copy of a doctoral thesis by

Steven R. Soltis

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Matthew T. O'Keefe

Name of Faculty Advisor(s)

A handwritten signature in dark ink, appearing to read "Matthew T. O'Keefe", is written over a horizontal line.

Signature of Faculty Advisor

August 12, 1997

Date

GRADUATE SCHOOL
Minneapolis, Minnesota

The Design and Implementation of a Distributed File System based on Shared Network Storage

Steven R. Soltis

Abstract

Distributed file systems allow users to access and share files from any computer connected to the distributed system. Distributed file systems typically do not achieve the same level of performance that local file systems provide due to the demands of resource sharing. For workloads with large storage capacity requirements, poor performance of distributed file systems often overshadows the benefits of transparent file sharing.

Traditional network and channel interfaces differ in performance, connectivity, and connection distance. By merging network and channel interfaces, resulting interfaces allow multiple computers to physically share storage devices. Computers service local file requests directly from network attached storage devices. Direct device access eliminates server machines as bottlenecks to performance and availability. Communication is unnecessary between computers, since each machine views storage as locally attached.

This dissertation presents a distributed file system design based on a shared network storage architecture. The architecture distributes user workloads and file system resources across the entire system. Functions once performed by server computers are redistributed to clients and storage devices. The design brings responsibilities, such as caching and consistency management, closer to hardware, so that these functions execute faster and more reliably.

The Global File System (GFS) is a distributed file system prototype built upon Fibre Channel networks. GFS is implemented in the Silicon Graphics IRIX operating system and is accessed using standard UNIX commands and utilities. GFS uses a consistency mechanism that is prototyped on Seagate disk drives and Ciprico disk arrays. This dissertation describes the architecture and implementation of the file system design. Performance analysis is given for the file system prototype in large data demand environments.

Acknowledgments

*I put my heart and soul into my work, and have half lost my mind
in the process.*

– Vincent Van Gogh

I wish to thank my committee, Matthew O’Keefe, David Lilja, Pen-Chung Yew, Larry Kinney, and Paul Woodward. In particular, I thank my advisor Matthew O’Keefe for his support and interest. Matt has been an avid proponent of the Global File System research.

Thomas Ruwart is the grandfather of the Global File System. Tom’s insight and vision significantly influenced GFS design. Grant Erickson wrote a substantial amount of GFS code. His contributions considerably enhanced GFS design and implementation. Thanks also goes to Kenneth Preslan, Benjamin Gribstad, Christopher Sabol, and Jonathan Brassow for their contributions and continuing efforts.

The SCSI Device Lock command was made possible because of contributions from Ciprico and Seagate Technology. Ciprico and Seagate also provided equipment and assistance necessary for GFS development and benchmarking. This work was supported by the National Aeronautics and Space Administration (NASA), the Office of Naval Research (ONR), and the National Science Foundation (NSF).

I thank the Laboratory for Computational Science and Engineering (LCSE) for tolerating many reboots and system crashes during GFS development. I especially thank Russell Cattelan and Joe Haberman for their indispensable assistance. I also thank the “Bear” cluster support staff who were essential during GFS benchmarking.

Graduate school has been fulfilling because of the close friendships that I have made with my fellow students and their spouses. Aaron Sawdey, the Oracle, has been a great source of knowledge and technical guidance. Peter Bergner taught me programming style and bike racing strategies. David Engebretsen, Corey Plender, Steven VanderWiel, and Kelvin Yue each answered my technical questions from time to time, though their important contributions were as lunch partners and fierce computer game opponents.

Finally, I wish to thank my family and friends for keeping me sane throughout my graduate school career. I am especially indebted to Katherine Fish. The quality of this dissertation is largely due to Kate’s meticulous editing.

Contents

Abstract	i
Acknowledgments	ii
List of Figures	viii
List of Tables	ix
1 Introduction	1
1.1 Problem Definition	2
1.2 Distributed File System Architecture Solution	3
1.3 Research Contributions	4
1.4 Dissertation Overview	4
2 General Background	6
2.1 Storage Devices	6
2.1.1 Disk Drives	8
2.1.2 Disk Arrays	10
2.1.3 Solid-State Storage Devices	11
2.2 I/O Interfaces	11
2.2.1 Channels	11
2.2.2 Networks	12
2.2.3 Merger of Channels and Networks	14
2.2.4 Network Attached Peripherals	14
2.3 Fibre Channel	14
2.3.1 Topologies	15
2.3.2 SCSI-3 and Network Attached Storage Devices	16
2.4 Layers of the UNIX Operating System	17
2.4.1 System Call Interface Layer	17

2.4.2	File System Layer	17
2.4.3	Buffer Cache Layer	18
2.4.4	Device Driver Layer	19
2.4.5	Layer Bypasses	19
2.4.6	Streams	20
2.5	File Systems	21
2.5.1	Files	21
2.5.2	Mount Points	21
2.5.3	Data Blocks	22
2.5.4	Superblocks and Free Lists	22
2.5.5	Inodes and Dinodes	22
2.5.6	Virtual File System (VFS)	23
3	Distributed File Systems	26
3.1	Distributed File System Terminology	26
3.1.1	Client-Server Model	26
3.1.2	Server State	27
3.1.3	Caching Schemes and Consistency	27
3.1.4	Simultaneous File Sharing	28
3.2	Network Attached Storage	28
3.2.1	NAS Architecture Taxonomy Summary	29
3.2.2	Analytic Model of Network Attached Storage	30
3.2.3	Model Analysis	35
3.3	Distributed File System Architectures	37
3.3.1	Network File System	38
3.3.2	Sprite File System	40
3.3.3	Andrew and Coda File Systems	40
3.3.4	Serverless Network File System	41
3.3.5	VAXcluster VMS File System	42

3.3.6	SIOF/HPSS File Systems	42
3.3.7	Cray Shared File System	43
3.3.8	IBM AIX Cluster File Systems	43
3.4	Discussion	44
4	GFS Architecture and Implementation	46
4.1	The GFS Architecture	46
4.1.1	Environment	48
4.1.2	Network Storage Pools	48
4.1.3	Memory Hierarchy	50
4.1.4	GFS Advantages	51
4.2	File System Structure	53
4.2.1	Network Storage Pool Design	53
4.2.2	Resource Groups	54
4.2.3	Superblock	54
4.2.4	Dinodes	54
4.3	Device Locks	57
4.3.1	Lock State Bits, Activity Bits, and Clocks	57
4.3.2	GFS Consistency and Device Locks	59
4.3.3	Client Failures	60
4.3.4	Device Failures	60
4.3.5	Deadlocks and Starvation	62
4.3.6	Comparisons with Existing Semaphore and Lock Mechanisms	62
4.4	File System Vnode Operations	64
4.5	Discussion	67
5	GFS Performance Analysis	68
5.1	Test Environment	69
5.2	Storage Subsystem Performance Characteristics	70
5.2.1	Device Lock Performance	71

5.2.2	Random Access Performance of Raw Devices	71
5.2.3	Sequential Access Performance of Raw Devices	75
5.3	GFS Performance	79
5.3.1	Single Client Performance	79
5.3.2	Multiple Client Aggregate Performance	88
5.3.3	Multiple Client Throughput Performance	95
5.3.4	Directory Performance	96
5.4	Summary	97
6	Conclusions	99
6.1	A Solution to Poor Distributed File System Performance	99
6.2	Future Directions	100
6.2.1	Cross-Platform Implementations	100
6.2.2	Additional Performance Benchmarking	101
6.2.3	Improving Small File Performance	102
6.2.4	Improving Scalability	102
6.2.5	Recovery Mechanisms	103
6.3	Summary	105
	Bibliography	106

List of Figures

2.1	Fibre Channel Topologies	16
2.2	Layers of the UNIX Storage and Network Subsystems	18
2.3	Traditional UNIX Dinode and Metadata Pointer Tree	24
3.1	Data Transfers within an NAS-01 System	33
3.2	Data Transfers within an NAS-2 System	33
3.3	NAS-01 Serial Data Transfer Time	34
3.4	NAS-01 Pipeline Data Transfer Time	34
3.5	NAS-2 Data Transfer Time	34
3.6	Model Analysis of Single Disk Characteristics	36
3.7	Model Analysis of Disk Array Characteristics	36
3.8	Distributed File System Architectures	38
4.1	GFS Distributed System	47
4.2	Exporting GFS to NFS and HTTP	49
4.3	NFS Control and Data Paths	52
4.4	GFS Control and Data Paths	52
4.5	Files Mapped onto Resource Groups and Subpools	55
4.6	GFS Dinode and Metadata Tree	56
5.1	GFS Test Configuration	70
5.2	Seagate Barracuda 9 Random Access Read Performance	73
5.3	Seagate Barracuda 9 Random Access Write Performance	73
5.4	Ciprico 7010 Random Access Read Performance	74
5.5	Ciprico 7010 Random Access Write Performance	74
5.6	Seagate Barracuda 9 Sequential 16 MB Read Performance	76
5.7	Seagate Barracuda 9 Sequential 16 MB Write Performance	76
5.8	Seagate Barracuda 9 Sequential 256 MB Read Performance	77
5.9	Seagate Barracuda 9 Sequential 256 MB Write Performance	77
5.10	Ciprico 7010 Sequential Read and Write Performance	78

5.11	Transfer Times of GFS on Disks for File Sizes between 0 Bytes to 4 MB .	81
5.12	Transfer Times of GFS on Disks for File Sizes between 0 Bytes to 128 MB	81
5.13	Transfer Times of GFS on Arrays for File Sizes between 0 Bytes to 16 MB	82
5.14	Transfer Times of GFS on Arrays for File Sizes between 0 Bytes to 256 MB	82
5.15	Transfer Rates of GFS on Disks for 4 MB Files	84
5.16	Transfer Rates of GFS on Disks for 16 MB Files	84
5.17	Transfer Rates of GFS on Arrays for 16 MB Files	85
5.18	Transfer Rates of GFS on Arrays for 256 MB Files	85
5.19	Test Configuration with a Dedicated Root Directory Device	89
5.20	Test Configuration with a Non-Dedicated Root Directory Device	89
5.21	Aggregate Transfer Rates for 16 MB Files with a Root Directory Device .	90
5.22	Scaled Speedups for 16 MB Files with a Root Directory Device	90
5.23	Aggregate Transfer Rates for 256 MB Files with a Root Directory Device	92
5.24	Scaled Speedups for 256 MB Files with a Root Directory Device	92
5.25	Aggregate Transfer Rates for 16 MB Files without a Root Directory Device	93
5.26	Scaled Speedups for 16 MB Files without a Root Directory Device	93
5.27	Aggregate Transfer Rates for 256 MB Files without a Root Directory Device	94
5.28	Scaled Speedups for 256 MB Files without a Root Directory Device	94
5.29	Throughput Scaling with Random Workload	96
5.30	Directory Performance	97

List of Tables

2.1	Seagate's Cheetah 10,000 RPM Disk Drives	9
3.1	Model Variable Definitions	31
3.2	Model Parameters	35
3.3	How NFS fits into the OSI Model	39
4.1	Device Lock Operations	58
4.2	Device Lock Example	61
4.3	System Call Mapping to GFS Functions	65
5.1	Raw Read vs GFS Read Performance	86
5.2	Raw Write vs GFS Create Performance	87
5.3	Raw Write vs GFS Write Performance	87

Chapter 1

Introduction

Well, sure, the Frinkiac-7 looks impressive ... But I predict that within 100 years computers will be twice as powerful, 10,000 times larger, and so expensive that only the five richest kings in Europe will own them.

– Professor John Frink, late 1970's

Since the creation of the Sun Microsystems Network File System (NFS) in 1985 [1], an enormous amount of research investigating distributed file system designs has taken place. The majority of this research focuses on distributing server functionality and improving caching strategies. Although superior designs exist, NFS remains the most popular distributed file system; descendants of the Carnegie Mellon University Andrew File System (AFS) [2] are a distant second. Predictions of Professor Frink,¹ given in the quote above, are just slight exaggerations with respect to distributed file system trends. Several distributed file system designs look impressive but are only slightly more powerful and seem at least 10,000 times larger and more complicated than NFS.

Traditional distributed file system designs rely heavily on software mechanisms for communication, consistency, and availability. To achieve respectable performance, traditional schemes require complex caching and consistency mechanisms. Due to technological advancements, new interface standards support network connections to storage devices. These network attached storage (NAS) devices enable distributed file system designs that are less complex and deliver higher performance than traditional techniques. These designs derive performance from direct communication between computers and devices. NAS devices

¹Professor John Frink is a character from the FOX television show *The Simpsons*.

actively manage caching and consistency. Furthermore, existing hardware redundancy methods facilitate high availability NAS devices. Distributed file system designs that use network attached storage devices achieve good performance without needless complexity.

Brian Kernigan once stated, "Controlling complexity is the essence of computer programming." Simplicity should be the goal of every design; complexity should be encouraged only as a means for job security.

1.1 Problem Definition

Distributed file systems provide mechanisms that allow users to access and share files from any computer connected to the distributed system. Distributed file systems typically do not achieve the same level of performance that local file systems provide due to the demands of resource sharing. For workloads with large storage capacity requirements, poor performance of distributed file systems often overshadows the benefits of transparent file sharing.

Large datasets are becoming more common. In 1991, Baker *et al.* repeated the 1985 University of California at Berkeley file system activity study [3][4]. Both studies measured file system activity from an office and engineering environment. Baker concludes that file sizes are more than an order of magnitude larger than in 1985. The study also reveals that large file transfers lead to activity bursts which stress file servers and caches.

Riedel and Gibson performed a similar study on Andrew File Systems at Carnegie Mellon University [5]. Riedel and Gibson verify observations of Baker by finding that bursts of activity overwhelm servers, even though these servers are idle close to 97% of the time. Ergo, file systems should be designed to handle demands much greater than normal activity in order to ensure reasonable response times during activity bursts.

File sizes from office and engineering environments are relatively small with respect to datasets from applications such as multimedia, scientific computing, and visualization [6]. As the performance of CPUs increases and memory and storage costs decrease, large datasets are becoming common. For example, Braham discusses the capacities of digital film editing. Academy film resolution is 2664 lines of 3656 pixels. With twelve bits devoted to each red, green, and blue color, one Academy frame requires 42 MB of digital storage. At twenty-four frames per second, one minute of digitized film occupies nearly 60 GB of

storage [7]. Since existing distributed file systems have difficulty supporting office and engineering environments, large datasets overwhelm these file systems.

Users typically restrict large datasets to local file systems, because existing distributed file systems cannot deliver adequate performance. For users to access files from various computers, users must explicitly copy files between machines. Copies may take place over high speed networks or with manual tape transfers via a "sneaker net." As Andrew Tanenbaum suggests, "Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway" [8].

Designs must develop distributed file systems to support existing workloads as well as loads not practical by today's standards. Say's Law of economics states that supply creates demand. However, even current demands surpass existing distributed file system capabilities. File system designers must look beyond traditional solutions, if next generation designs are to support current and future requirements.

1.2 Distributed File System Architecture Solution

Traditional network and channel interfaces differ in performance, connectivity, and connection distance. By merging network and channel interfaces, resulting interfaces allow multiple computers to physically share storage devices. Computers service local file requests directly from network attached storage devices. Direct device access eliminates server machines as bottlenecks to performance and availability. Communication is unnecessary between computers, since each machine views storage as locally attached.

This dissertation presents a distributed file system design based on a shared network storage architecture. The architecture distributes user workloads and file system resources across the entire system. Functions once performed by server computers are redistributed to clients and storage devices. The design brings responsibilities, such as caching and consistency management, closer to hardware, so that these functions execute faster and more reliably.

The Global File System (GFS) is a distributed file system prototype built upon Fibre Channel networks. GFS is implemented in the Silicon Graphics IRIX operating system and is accessed using standard UNIX commands and utilities. GFS uses a consistency mechanism that is prototyped on Seagate disk drives and Ciprico disk arrays. This dissertation describes the architecture and implementation of the file system design. Performance analysis is given for the file system prototype in large data demand environments.

1.3 Research Contributions

The main contributions of this dissertation and research include the following:

- A distributed file system design that uses private file managers and network attached storage devices was developed. The design distributes work and resources across the entire distributed system. Clients and storage devices execute operations that are traditionally performed by server computers. Unlike similar designs, this architecture allocates file system resources across pools of storage devices.
- A distributed file system classification is presented that builds upon the network attached storage (NAS) taxonomy defined by Gibson *et al.* [9]. This classification scheme characterizes distributed file systems based upon file manager organization and the NAS taxonomy. Such characterizations illustrate differences between file system designs.
- The Global File System design is implemented on a real operating system. This implementation is necessary for performance benchmarking, and in time, could be a useful tool. Performance measurements are given to provide insight that would not be possible using analytic modeling or simulation. Unlike similar designs that parallelize local file systems, GFS is designed and written from scratch as a distributed file system.
- A file system consistency mechanism was defined and developed. This mechanism, managed by network attached storage devices, is a simple, yet elegant, coherence scheme. The mechanism supports mutual exclusion, caching, and failure recovery. Availability is achieved using RAID-like technologies. Other device managed consistency mechanisms, like SCSI Reserve and Release [10], are limited in granularity and adversely affect performance of other device commands. Computer-managed coherence schemes have complex designs that require large amounts of network traffic.

1.4 Dissertation Overview

This dissertation consists of six chapters. Chapter 1 presents an argument for further research in distributed file systems; the remainder describes a design that addresses performance problems common to traditional designs.

Chapter 2 reviews some key components upon which distributed file systems are built. Distributed file system designs are influenced by almost every component of computer systems. These components include hardware, such as storage devices, memory systems, networks, and processors, in addition to multiple levels of operating system software. Chapter 2 defines terminology for a general audience.

Chapter 3 examines distributed file systems. This chapter introduces rudimentary concepts and summarizes important distributed file system designs. Distributed file system designs are categorized by file manager organization and network attached storage architecture. A first order analytic model demonstrates performance benefits of network attached storage.

Chapter 4 discusses architecture and implementation of the Global File System. This chapter presents GFS design and describes advantages over competing architectures. A prototype description details file system layout, network storage pool layer design, device locks used for data coherence, and important file system functions.

Chapter 5 explores the performance of the file system. Comparisons are made between GFS and existing file systems. Analysis primarily focuses on comparing GFS to raw storage subsystem performance. Several different metrics are used to accurately characterize GFS performance.

Chapter 6 summarizes benefits of the Global File System design. This chapter also introduces areas that would benefit from continued research.

Chapter 2

General Background

Nothing is particularly hard if you divide it into small jobs.

– Henry Ford

Computer system components influence distributed file system designs. These components include storage devices, networks, memory systems, processors, and multiple levels of the operating system. This chapter provides background of each component and defines terminology for a general audience. This review emphasizes system components that are important to later discussions of the Global File System. Chapter 3 presents background specific to distributed file systems.

2.1 Storage Devices

Storage device characteristics that impact file system design are access times, transfer rates, addressability, cost, capacity, and availability. Given these traits, file system designers make tradeoffs during implementation.

Storage device performance influences how file systems access data. File systems often divide user data requests into several storage device requests. Equation 2.1 linearly models file system request transfer times (T_r). Request transfer times consist of startup times (T_s) and data transfer times (T_d). Startup times depend upon attributes of the storage subsystem, including device characteristics and current device states. For this analysis, assume startup times are nearly constant. Request transfer time is a function of request size (S) and the storage subsystem transfer rate (R).

Equation 2.2 expresses total user transfer time (T_t) as the summation of all file system request times. Suppose that T_s equals 10 ms and R equals 10 MB/s. The request time for 2KB of data would be $10\text{ms} + (2\text{KB})/(10\text{MB/s}) \approx 10.2\text{ms}$; total transfer time for 4KB of data with two 2KB requests would be 20.4ms. The request time for one 4KB request would be $10\text{ms} + (4\text{KB})/(10\text{MB/s}) \approx 10.4\text{ms}$. In this example, two 2KB transfers take nearly twice as long as one 4KB transfer, since T_s dominates the S/R term. In general, total transfer time for a fixed amount of data is a function of the number of requests.

$$T_r = T_s + T_d = T_s + S/R \quad (2.1)$$

$$T_t = \sum T_r \quad (2.2)$$

Storage device addressability is another file system issue. Devices are block addressable while system memory is byte addressable. Mapping between system memory address space and storage device address space is, in part, the responsibility of a file system. Poor mapping may require file systems to issue multiple requests to storage devices.

Cost and capacity are two loosely related factors that influence file system data management strategies. A typical ratio employed by system designers is dollars per megabyte. Either a fixed cost determines capacity or a minimum capacity dictates cost. This ratio applies to both storage and memory.

Components that are unavailable, either failed or off-line, may cause file system data to be inaccessible or even lost. Redundancy schemes improve availability. However, coherency mechanisms that maintain consistency between copies of data often reduce performance and increase system complexity.

Through various compromises, well-engineered file system designs balance device characteristics. Given that storage device traits independently change with time, file system designs must update periodically. In the past, storage was expensive, so designers attempted to efficiently use storage in order to minimize costs. Due to a significant decrease in storage costs, file system designs currently focus on optimizations such as sacrificing storage space efficiency to increase performance. Several similar tradeoffs are detailed later in this chapter.

2.1.1 Disk Drives

Disk drives are non-volatile storage devices that record data on magnetic media [11][12]. Each disk device consists of one or more stacked platters attached to a rotating spindle. Disks magnetically store data on recording surfaces located on both sides of each platter.

Concentric circles called tracks divide each platter surface. Sectors divide tracks into the smallest unit that can be read or written, typically a constant size of 512 bytes. A sector is also referred to as a disk block.

Early disk drives allotted a constant number of sectors per track, so the recording densities were lower for outer tracks than inner tracks. Today, recording densities are relatively constant for all tracks. Zones of adjacent tracks divide the recording surface. Zones consisting of outer tracks have more sectors than inner track zones. This technique is known as zone bit recording.

Each recording surface has one head that reads and writes sector data. Heads attach to the ends of actuator arms. These arms pivot the heads, in unison, between tracks. Seek time is the duration required to position a head to the desired track. Rotational latency is the time spent after a head seek but before the target sector rotates under the head. Together, seek time and rotational latency comprise total positioning time of a request.

Modern devices possess speed-matching buffers. To coordinate between channel availability and media latencies, devices temporarily store data in speed-matching buffers. Some advanced devices use buffer memory for caching. Caches may act as read-ahead buffers by prefetching likely to be read data. Caches may also write buffer requests by transferring data into buffers and then releasing the channel. Devices write cached data to media at a later time. Advanced configurations function as both read and write caches. Read and write caches typically employ variants of least recently used (LRU) replacement policies [13].

Disks that support command queuing maintain lists of outstanding requests. Disks may reschedule requests in order to reduce seek times. Reordering often increases device throughput.

Grochowski and Hoyt predict the status of disk technologies for the year 2000 [14]. Given areal densities increase at an annual compound growth rate of 60%, a 3.5 inch disk could store up to 90 GB. This density trend leads to a decrease in dollars per MB; prices at the end of the decade may be well under \$0.01/MB. If transfer rates continue to increase at

about 40% per year, 50 MB/s will be possible by 2000. Rotation speeds and seek times, however, continue to improve at much slower rates.

As of 1997, Cheetah disk drives from Seagate are state-of-the-art. Table 2.1 summarizes low and half-height profile Cheetah disk statistics. At 10,000 RPM, these 3.5-inch disks are currently the fastest available [15][16].

Vender	Seagate	Seagate
Model	ST34501FC	ST19101FC
Capacity	4.55 GB	9.10 GB
Interface	Fibre Channel	Fibre Channel
Form-Factor	3.5 inches	3.5 inches
Profile	Low	Half-high
Sector Size	512 bytes	512 bytes
Spindle Speed	10,000 RPM	10,000 RPM
Recording Surfaces	4	8
Buffer Size	512 to 2048 KB	512 to 2048 KB
Transfer Rate	11.3 to 16.8 MB/s	11.3 to 16.8 MB/s
Seek Times		
Read Track-to-Track	0.78 ms	0.8 ms
Write Track-to-Track	0.85 ms	0.9 ms
Max	18 ms	18 ms
Max Latency	6.00 ms	6.00 ms

Table 2.1: Seagate's Cheetah 10,000 RPM Disk Drives

Technology trends significantly affect system architectures and applications. For instance, average latency of a Cheetah disk is half the maximum latency. Ignoring seek time, T_s equals 3 ms. With outer track transfer rates of 16 MB/s, a 4 KB transfer request takes 3.25 ms. Startup time dominates this duration, especially for small transfers. Currently, slow seek times require architectures to include large, elaborate caches on systems and, to a lesser extent, devices. Caching is the conventional technique to compensate for large access times. Though memory costs continue to decrease at a substantial rate, most file systems do not have the necessary locality to exploit larger caches. Prefetching is another technique to reduce access times. Sequential accesses benefit from prefetching, though many file system requests are discontinuous and non-sequential.

2.1.2 Disk Arrays

Disk arrays are storage devices designed for performance and availability [17][18]. Disk arrays consist of several single disks managed by controller hardware. The controller distributes data across all devices such that the array appears as a large-capacity device.

Disk arrays deliver transfer rates faster than single disks. Arrays stripe data across disks by assigning consecutive block addresses to each successive disk. Given n striped disks and large requests, the transfer rate is approximately n times greater than that of a single disk. However, access times of disk arrays are not necessarily faster than those of single disks. Depending upon array configuration and request alignment, data transfer might not begin until all disks complete individual requests.

Disk arrays provide higher availability than single disks. Disk arrays fully or partially replicate data. Given single disk failure, arrays function without data loss. Some disk arrays rebuild lost data without taking the array off-line. Redundant components, such as power supplies, achieve even higher availability.

The taxonomy of Redundant Arrays of Inexpensive Disks (RAID) classifies disk arrays [19]. Briefly, RAID 0 refers to a configuration with block-level striping and no redundancy. Write performance is high, since no redundant information needs updating. RAID 1, called mirroring, stores two copies of the data. Storage efficiency is poor but read and write performance remains high. RAID 2 uses a memory style error-correction-code (ECC) of redundancy. Storage efficiency of RAID 2 depends upon the number of data disks. Increasing data disks logarithmically increases redundancy disks. RAID 3 interleaves data at the byte level and places parity on a single disk. RAID 3 hardware controllers provide performance suitable for high-bandwidth applications. RAID 4 is similar to RAID 3, except the RAID controller interleaves data at the block level. RAID 5 interleaves at the block level and distributes parity blocks across all disks. RAID 5 provides high data efficiency and good overall performance.

These five array configurations have different performance characteristics. Performance is dictated by device access patterns, parity layout, and implementation. For instance, distributed parity allows RAID 5 devices to process multiple requests in parallel, where as parity disks of RAID 3 and RAID 4 configurations become hot-spots of activity. However, RAID 3 devices often possess hardware controllers that deliver higher bandwidth performance than software RAID 5 designs.

2.1.3 Solid-State Storage Devices

Solid-state storage devices (SSD) are block addressable devices that electrically store data in battery powered DRAM memories. Solid-state storage devices have significantly lower latencies and higher transfer rates than disks and disk arrays. Price per storage of solid-state devices is at least an order of magnitude higher than magnetic counterparts. Capacity per unit volume is typically lower for SSD than magnetic disks. Solid-state devices are useful for applications requiring higher performance than magnetic storage.

2.2 I/O Interfaces

I/O interfaces transport data between computers and devices. Interfaces often become open standards. Traditionally, interfaces fall into two categories: channels and networks. Current trends merge channels and networks into single interfaces.

2.2.1 Channels

Channel interfaces predictably transfer data with low-latency and high-bandwidth performance. High-performance requirements often dictate that hardware mechanisms control channel operations; even though, hardware design tends to lower channel flexibility. Channels span short distances and provide low connectivity – typically less than 25 meters and 16 devices, respectively.

Small Computer System Interface (SCSI) is a popular channel interface that the American National Standards Institute (ANSI) defines as standard X3T9.2 [10][20][21]. SCSI connects host computers and storage devices via 10 MB/s, 20 MB/s, and 40 MB/s shared buses. SCSI controllers translate between logical block addresses and physical sectors; devices recover from media errors by re-mapping bad sectors. SCSI devices buffer requests in order to match data transfer speeds between media and channels.

In the past, channel designs tightly integrated protocol layers with physical layers; this is changing. For instance, SCSI-2 pairs a command protocol with a parallel bus. SCSI-3 separates an enhanced SCSI-2 command set from the physical level, therefore allowing the command set implementation to run on any physical channel. Intelligent Peripheral Interface level 2 (IPI-2) is a parallel bus implementation; IPI-3 maps to several physical interfaces including High Performance Parallel Interface (HiPPI).

The SCSI protocol defines several commands and status messages. Two basic SCSI disk commands, `read` and `write`, specify starting block addresses as well as the number of blocks to transfer. Initiators send SCSI commands to target devices. SCSI does not preclude devices from acting as initiators or host adapters from acting as targets. Host adapters transfer data between system memory and devices via direct memory access (DMA). DMA eliminates CPU involvement during data transfers between host adapters and memory.

Each SCSI command returns status information to initiators. Two common status messages are `good` and `check condition`. Good status indicates successful command completion. A check condition message reveals an error or failure. After issuing a check condition, devices retain information describing the problem. Initiators read this information, called sense data, in order to perform recovery.

2.2.2 Networks

Networks are interfaces with more flexibility than channels. Networks have high connectivity, connect long distances, and operate in unpredictable environments. Networks typically separate protocol layers from physical layers. Software controls a great deal of network functionality, providing networks with flexibility but low performance [22].

Designs organize networks into layers. Layers interact with adjacent layers through well specified interfaces. Layers use the facilities of lower layers to communicate with corresponding layers on other machines [8].

The International Standards Organization (ISO) defines a reference model called the Open Systems Interconnection Reference Model, or simply the OSI model [23]. The OSI model describes a set of layers for network protocols, but does not specify the implementation of each layer. The OSI model is not a protocol, just an outline consisting of seven layers:

- | | |
|----------------------|--|
| 1 - Physical | Transmits bits of data over physical medium. |
| 2 - Data Link | Controls access to medium. Groups bits into frames that are free of transmission errors and sends back frame acknowledgments. |
| 3 - Network | Groups frames into packets and routes packets appropriately. Since packets may have no relation to other packets, communication is connectionless. |

- 4 - Transport** Provides end-to-end communication between systems. Transport services may include flow control, message ordering, and reception acknowledgments.
- 5 - Session** Establishes sessions between remote processes that support enhancements beyond the transport layer.
- 6 - Presentation** Describes semantics of transferred data, such as integer and ASCII character data types.
- 7 - Application** Programs and functions that require communication.

Networks are either shared media or point-to-point connections. Shared media networks connect multiple systems via shared buses. Systems arbitrate for the media using token or collision detection schemes. Point-to-point networks connect systems and switching devices by means of dedicated network links. Switches store and route messages to appropriate paths.

Local area networks (LAN) connect machines short distances, such as within buildings. Wide area networks (WAN) span longer distances, like across campuses or even across the world. LANs normally consist of shared media networks while WANs are point-to-point connections [8].

The Institute of Electrical and Electronics Engineers (IEEE) developed a set of LAN standards consisting of Carrier Sense Multiple Access with Collision Detection (CSMA/CD), token bus, and token ring [24][25][26]. The CSMA/CD standard, numbered 802.3, is a popular collision-based network. Ethernet, a name borrowed from a Xerox product, is the common name for the 802.3 network standard. The 802.3 standard provides different interconnect speeds; 10 Mbit/s is the most popular speed. When 802.3 was designed in 1978, 10 Mbit/s was considered very fast. Over time, 10 Mbit/s became a bottleneck in LAN environments. 100 Mbit/s and 1 Gbit/s Ethernet standards currently exist.

Point-to-Point WANs play a significant role in LANs. Asynchronous Transfer Mode (ATM) is such a network. Telecommunication companies designed ATM as a switchable WAN, but LAN environments also use ATM. While still an order-of-magnitude more expensive than Ethernet, ATM can connect high-end workstations or act as LAN backbones by connecting Ethernets via routers.

Ethernet and ATM only support physical and data link OSI layers. As a result, these networks require support from higher layer protocols like Transmission Control Protocol

(TCP) and Internet Protocol (IP). IP, a network layered protocol, provides routing based on 4-byte addresses [27]. TCP, a transport layer protocol, facilitates connection-oriented communication with guaranteed, in-order delivery [28]. User Datagram Protocol (UDP) is a connectionless transport layer protocol without guaranteed delivery [29]. UDP could replace TCP provided that applications handle error recovery.

2.2.3 Merger of Channels and Networks

Recent interface trends combine channel and network technologies into single interfaces capable of supporting multiple protocols [30][31]. Interface merging tends to produce slightly more complicated designs, but these interfaces generally inherit advantages of both channels and networks. Combining these traditionally independent subsystems enables vendors to produce single products with multiple uses. Vendors providing combined products benefit from larger markets.

2.2.4 Network Attached Peripherals

A network attached peripheral (NAP), according to Van Meter, is “a network computer peripheral that communicates via a network rather than a traditional I/O bus, such as SCSI” [32]. Van Meter presents characteristics that distinguish NAPs from bus based devices. Such characteristics include interconnect distance, ownership, and an ability to handle general network traffic. Printers and computer terminals often fit this definition. Other peripherals like scanners and storage devices may also be designed as NAPs.

Merging channels and networks provides new functionality to devices. Network attached storage (NAS) devices connect directly to networks. Multiple computers can share NAS devices. Chapter 3 summarizes an NAS taxonomy as defined by Gibson *et al.* This taxonomy partially characterizes distributed file system designs. Storage Area Networks (SAN) are dedicated networks that connect NAS devices to small computer clusters.

2.3 Fibre Channel

Fibre Channel (FC) is an emerging ANSI serial interface that supports channel and network operations [33][34][35]. Fibre Channel consists of five functional levels. Modular design

allows independent implementation of each level. Fibre Channel defines levels FC-0 through FC-4:

- FC-4: Protocol mapping** defines the mapping of protocols to lower levels. FC-4 supports multiple protocols such as SCSI, IPI, HiPPI, 802.2, IP, and ATM.
- FC-3: Common services** define services that span multiple ports. Such services include striping and multicasting.
- FC-2: Framing protocol/flow control** defines framing, signaling, and flow control. FC-2 supports point-to-point, arbitrated loop, and switched fabric topologies.
- FC-1: Encode/decode** defines the transmission protocol that uses an 8B/10B encoding scheme.
- FC-0: Physical mapping** defines physical media and data rates. FC-0 supports 1.062 Gbit/s electrical and optical connectors and media. Optical media includes lasers and Light Emitting Diodes (LED); copper media includes coax and twisted pair. FC-0 permits higher data rates.

2.3.1 Topologies

Fibre Channel defines all computers, switching elements, and storage devices as nodes. Each node has one or more FC ports called network ports (N_Ports). Each port possesses a transmitter and a receiver to interface media. Fibre Channel supports three topologies illustrated in Figure 2.1: point-to-point, arbitrated loop (FC-AL), and fabric. Point-to-point topology simply connects two N_Ports.

Like shared buses, only one arbitrated loop node transmits at any given time. As many as 126 nodes connect together to form a ring; the transmitter of one node connects to the receiver of another. A transmitting node must first arbitrate for the loop. After acquiring the loop, the transmitting node either sends messages to other nodes or broadcasts to several nodes. After transmission, the node releases the loop. Loop ports (L_Ports) are nodes that support the arbitrated loop topology.

Fabric topology dynamically connects nodes in a switched network. Switching elements contain fabric ports (F_Ports). F_Ports connect to N_Ports and other F_Port switching elements. F_Ports may connect to L_Ports on arbitrated loops. Unlike arbitrated loops that

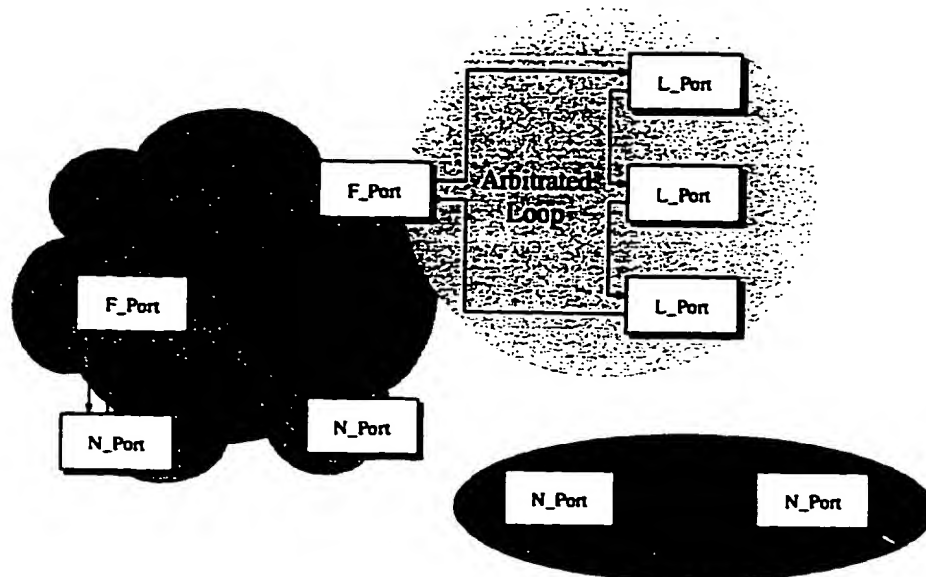


Figure 2.1: **Fibre Channel Topologies.** Point-to-point configurations are the simplest. This topology connects node pairs or acts as backbones for other network types. Arbitrated loops connect several ports via shared media. Fabrics connect ports to switched environments.

share media bandwidth, fabric bandwidth scales with the addition of ports and network links.

Performance and cost requirements dictate network topology. Many nodes may attach to inexpensive, arbitrated loops in order to share loop bandwidth and connectivity. Nodes can also attach to fabric ports and exploit the network bandwidth. Loops connect to fabric ports for high connectivity but relatively low costs.

2.3.2 SCSI-3 and Network Attached Storage Devices

FC-4 supports the SCSI-3 command protocol. Given Fibre Channel support for SCSI, multiple computers may directly share storage devices attached to the network. Several devices can share the bandwidth of arbitrated loops or devices may connect through scalable bandwidth fabrics.

Because each F_Port switching element roughly costs the same as a FC disk, direct connection of FC disks to fabrics is currently a poor economic solution. Loops provide an inexpensive solution with high connectivity and good performance. Fabric attachment is

useful for high bandwidth disk arrays that would otherwise saturate single loops. Fabric attached loops provide a balanced cost and performance configuration.

2.4 Layers of the UNIX Operating System

This section presents operating system background with respect to UNIX operating system layers. Reasons for presenting UNIX include: (1) variations of UNIX run on platforms ranging from personal computers to supercomputers, (2) a large amount of UNIX design literature and research exists, (3) numerous non-UNIX operating systems inherit UNIX design principles, and (4) the GFS prototype targets the Silicon Graphic IRIX operating system which is a UNIX variant.

UNIX functionally organizes storage and network subsystems into layers [36]. Figure 2.2 illustrates these layers. Each layer views storage and communication through different degrees of abstractions. The top layer of Figure 2.2 is user space. Programs operating at this level include command shells and system utilities; applications run on top of these programs. All user level programs interact with the operating system, or kernel, through system calls.

2.4.1 System Call Interface Layer

The system call interface is the well-defined boundary between user and kernel space. System calls are operating system routines that perform tasks for users, such as allocating memory, accessing storage devices, and creating new processes. Numerous system call routines exist including `open`, `close`, `read`, `write`, `creat`, `unlink`, and `readdir`.

2.4.2 File System Layer

The file system layer lies beneath the system call layer. Modern UNIX implementations include installable file system interfaces. Many UNIX implementations incorporate the Virtual File System (VFS) interface. VFS, developed by Sun Microsystems, provides a common interface to file systems. VFS divides file system functionality into file system and individual file operations [1][37].

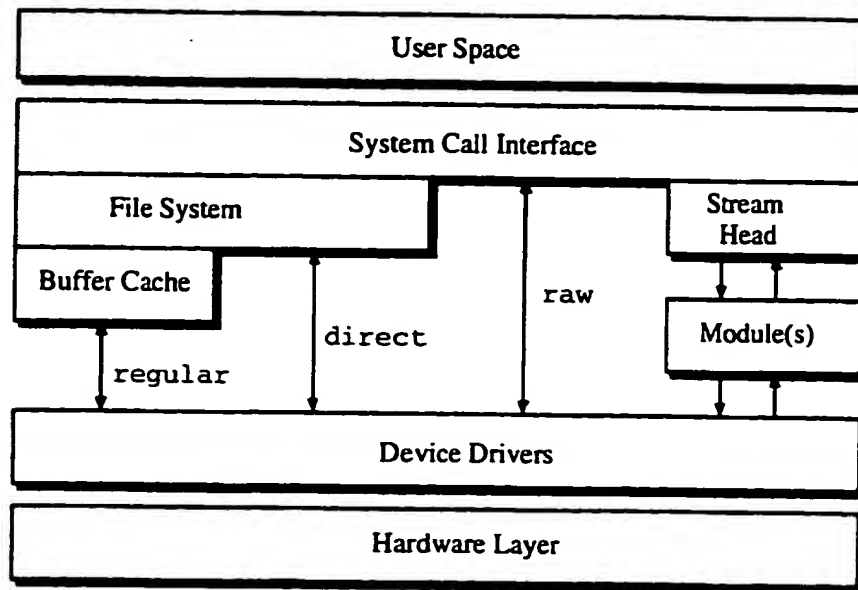


Figure 2.2: **Layers of the UNIX Storage and Network Subsystems.** This figure illustrates logical layers of the UNIX I/O system. The highest abstractions occur at user space. In contrast, the device driver layer is the most hardware specific.

2.4.3 Buffer Cache Layer

File systems often use buffer cache services. The buffer cache layer consists of system memory buffers and routines that operate on these buffers. Buffer caches provide caching, prefetching, and temporary memory for non-aligned transfers. These caches reduce device read and write requests by caching recently accessed data. Buffer caches either write-through or write-behind, and often use a least recently used (LRU) replacement policy. Other replacement algorithms are random replacement (RR) and least frequently used (LFU) policies [13][38]. File systems prefetch data into buffer caches for possible future reference. Prefetches are often continuations of current requests. Such prefetching only increases data transfer times of request durations, since devices already perform initial seeks and rotations for the non-prefetched data. File systems also use buffer caches to temporarily store data for non-aligned transfers.

2.4.4 Device Driver Layer

The device driver layer is the lowest software level. Device drivers, or drivers, are interfaces between the kernel and hardware. Drivers hide hardware device specifics. Several levels of device drivers comprise the device driver layer. High-level device drivers are more abstract than low-level drivers. Low-level drivers are most hardware specific [39].

I/O requests arrive at device drivers from higher in the kernel. Drivers use input requests to construct requests suitable for lower drivers or hardware. Drivers place newly formed requests on the driver queue. These queues are usually first-in, first-out (FIFO), but can be priority-based to enable scheduling. Drivers pass queued requests down to lower device drivers or the hardware level.

Modern devices generate interrupts to eliminate the burden of systems constantly checking request statuses. Processes issue requests and then sleep until woken by interrupt handlers. Interrupt handles are device driver routines. The kernel calls interrupt handlers when devices or other drivers complete requests. Interrupt mechanisms identify completed requests and call the appropriate handler. Drivers remove requests from the queues, update request statuses, and pass requests to the next higher interrupt handlers. Since handlers interrupt execution of unrelated processes, these routines must communicate with sleeping processes by using mechanisms like semaphores.

2.4.5 Layer Bypasses

For performance reasons, UNIX includes layer bypasses. Figure 2.2 illustrates paths between upper layers of the operating system and device driver layers. From left to right, these paths are regular, direct, and raw I/O. Regular I/O is the most common path, since users frequently store data on file systems. Most file systems use buffer caches to enhance performance and provide a mechanism to access non-block-aligned objects. Cache misses, however, require two data transfers: one between the device and buffer cache and one between the buffer cache and user memory. For applications with limited or no locality of reference, buffer caches require an additional memory copy in the data path and additional overheads for cache management.

Raw I/O is the other extreme. Users perform data management functions without aid from file systems. Raw I/O bypasses much of the operating system, providing users with a very efficient method of accessing data. Devices directly transfer data to and from user memory, eliminating copies between the buffer cache and user memory space.

Direct I/O is a compromise between raw and regular I/O; the direct path includes file systems but bypasses buffer caches. For applications where caching is not useful, direct I/O has advantages of both regular and raw I/O. The file system facilitates data management, but direct I/O eliminates the memory copy between the buffer cache and user memory. Not all UNIX platforms implement direct I/O.

2.4.6 Streams

Streams are full-duplex data transfer paths between user level processes and network device drivers. STREAMS is the modular mechanism for providing network and communication services to applications. STREAMS consists of system calls, stream heads, modules, and network device drivers [40]. Network device drivers and hardware fit into the OSI model at the data link level. These drivers are specific to network interface hardware.

Applications access stream heads through the system call interface. Once opened, processes write data to streams using either `write` or `putmsg` system calls. These system calls copy data from user address space to kernel space. Processes read data from streams using either `read` or `getmsg` system calls. Processes block until messages are available at the heads and then copy data from kernel to user space.

Modules are optional components within stream data paths. Each module accepts messages, processes messages, and passes new messages to the next level. Modules fit into the OSI model at session, transport, and network levels.

A Remote Procedure Call (RPC) is a procedural interface that provides an abstract view of communication. Clients communicate with server computers by invoking RPCs on servers. Servers accept commands and parameters from clients and execute the routines locally. Synchronous client processes sleep until functions complete. Once completed, servers return results to clients. RPC fits into the session layer of the OSI model.

RPC uses the External Data Representation (XDR) standard to describe data independent of machine architecture [41]. XDR encodes high-level language types by resolving issues such as type size, big-endian and little-endian byte ordering, and alignment within complex data structures. Context dictates data types, since encodings are implicit and do not identify type. XDR fits into the presentation layer of the OSI model.

2.5 File Systems

File systems manage user and system data on secondary storage. Applications often address data at the byte level, though storage devices are typically block addressable. File systems perform translations between byte and block level addressing schemes.

The terms metadata and real data classify file system structure data and user data, respectively. In other words, real data is data that users store in files. File systems create metadata to store layout information; metadata is not directly visible to users. The next few sections describe metadata structures.

2.5.1 Files

Files abstractly hide details concerning storage management from users. UNIX recognizes several file types. File systems and application programs handle each file type differently. Users store and retrieve data from regular files as contiguous, randomly accessible segments of bytes. Users are responsible for organizing data stored in regular files.

Directories are file abstractions that organize collections of files. In most modern file systems, directories nest within one another, thereby forming a tree structure with empty directories and regular files at the leaves. File names are unique to directories but not to file systems. Applications identify files by complete pathnames. A file name and the names of all encompassing directories comprise a pathname. The entire collection of file names composes the file system name-space.

2.5.2 Mount Points

Computers may simultaneously access files from multiple file systems. To provide a transparent name-space, file systems mount other file systems at mount points. Users transparently traverse from one file system, across a mount point, to another file system. File system root directories attach to mount points.

2.5.3 Data Blocks

File systems transfer blocks of data between system memory and storage devices. File system block sizes are multiples of device block sizes. File system block sizes are typically 4 KB to 16 KB.

Internal fragmentation occurs when file systems do not fill entire blocks. Traditionally, file systems limited internal fragmentation by using small file system block sizes. Today, conservation of storage is less important, so optimizations focus on improving file transfer rates. External fragmentation is due to non-contiguous storage of data blocks. Since I/O requests incur substantial device overheads, external fragmentation strongly influences transfer rates. Increases to file system block sizes tend to reduce external fragmentation, however large block sizes increase internal fragmentation.

2.5.4 Superblocks and Free Lists

File system superblocks contain information relating to file system structures and states. Superblocks maintain information concerning the amount of free space left on the file system, the device on which the file system is mounted, and file system access privileges. Superblocks also maintain pointers to locate file system root directories.

Additionally, file systems maintain free lists of unallocated data blocks. Most modern file systems manage free lists by means of bitmap tables. File systems set bits to signify blocks that are allocated to files.

2.5.5 Inodes and Dinodes

Every file has a corresponding disk index node, called a dinode. Dinodes store ownership information, access permissions, access times, and file sizes. File systems read dinodes into system memory structures called index nodes, or inodes. In literature, the term inode often describes both inodes and dinodes, since these structures are identical in many file systems. For clarity, a distinction is made between the two; inodes are in-core data structures while dinodes are structures stored on devices.

Directory files contain file names and unique numbers, called inode numbers. File systems use inode numbers to locate dinodes stored on disk. The separation of inode numbers from

file names allows file systems to support link files. Link files transparently reference other files. Links provide multiple file names for single files.

Dinodes are either dynamic or static. File systems create dynamic dinodes during file allocation. File system utilities create static dinodes while building new file systems. File system designs typically allocate statically, though some systems may combine both allocation schemes.

Dynamic dinode allocation has the benefit that dinodes only exist when corresponding files exist. With static allocation, file system creation utilities must overestimate the maximum number of files the file system will contain; conservative estimations waste storage space. With dynamic allocation, users cannot deplete dinodes before exhausting free data blocks. The advantage of static dinode allocation is a trivial inode number to dinode device address translation. Dynamic allocation must perform this translation through lookup tables.

Dinodes maintain data block addressing information. Dinodes store lists of pointers; each pointer addresses a data block. These blocks either contain real data or lists of other pointers. Tree structures form with dinodes at the roots and real data blocks at the leaves.

Figure 2.3 illustrates the structure of a dinode and metadata pointer tree for a traditional UNIX file system. The first ten pointers are direct pointers that address data blocks. The first indirect pointer addresses a data block filled with direct pointers. The second indirect pointer addresses a data block filled with indirect pointers. The third indirect pointer addresses a block of double indirect pointers, taking a total of three indirections to reach real data.

Extent-based addressing is another approach to dinode pointers. An extent-based pointer contains the first block address of the extent and the number of blocks in that extent. Dinode trees need fewer pointers, because each pointer addresses several consecutive blocks. Unfortunately, extent addressing produces irregular tree structures that are complicated to allocate, traverse, and defragment.

2.5.6 Virtual File System (VFS)

The virtual file system (VFS) is an interface that supports various file system types within a kernel. Several UNIX implementations incorporate VFS; however, interfaces differ from one platform to another. Like most UNIX implementations, VFS is similar across platforms but far from compatible [42][43].

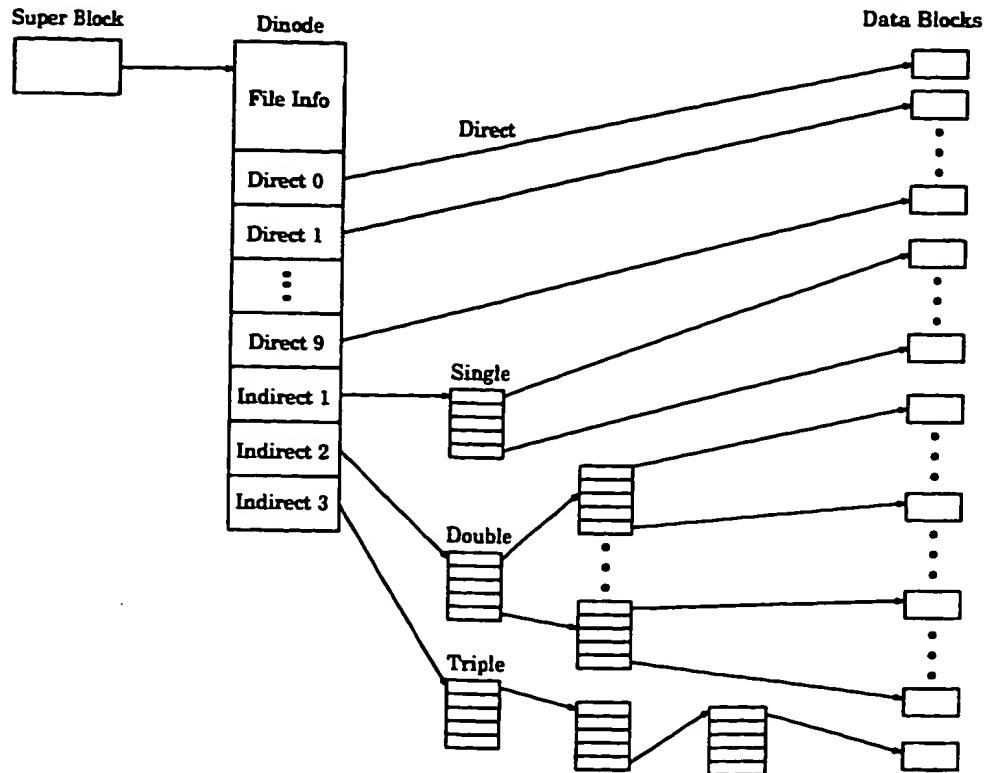


Figure 2.3: **Traditional UNIX Dinode and Metadata Pointer Tree.** The UNIX dinode contains header information and several pointers. The first ten pointers directly address data blocks. The first indirect pointer addresses a block consisting of direct pointers. The second indirect pointer addresses a block of indirect pointers. This scheme continues with a triple indirection to form an irregular tree structure.

VFS is an object-oriented interface. This interface defines virtual VFS and vnode operations. Each installed file system provides the kernel with functions associated with VFS and vnode operations. VFS operations include `VFS_MOUNT`, `VFS_UNMOUNT`, and `VFS_STATVFS`. These functions operate on file systems by mounting, unmounting, and reading status, respectively.

Vnode operations manipulate individual files. A vnode is the VFS virtual equivalent of an inode. VFS creates and passes vnodes to file system vnode operations. Vnode operations include opening, closing, creating, removing, reading, writing, and renaming files. VFS defines many other vnode operations, yet file system implementations need only support a subset of these routines.

To open a file, VFS passes the file pathname to the vnode operation, `VOP_LOOKUP`. `VOP_LOOKUP` maps to a file system specific lookup function. This function searches the name-space for each qualifier in the pathname. If this lookup function comes across a mount point to another file system, the lookup routine calls the vnode lookup operation corresponding to this mounted file system. Eventually, the lookup routine finds the file and returns the file vnode to the caller.

The vnode operation, `VOP_CREATE`, creates new files. Actions of this operation vary between file systems, but the common case allocates dinodes and writes file names to directories. Read and write system calls invoke `VOP_READ` and `VOP_WRITE` operations. `VOP_READ` and `VOP_WRITE` call `VOP_BMAP` routines. `VOP_BMAP` translates file byte offsets to block numbers. Read and write vnode operations use the byte to block translations during file data transfers.

Chapter 3

Distributed File Systems

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

– Leslie Lamport, 1992

This chapter reviews distributed file system background, introduces a distributed file system classification, and summarizes related research. The classification of distributed file system architectures characterizes file manager organizations and builds upon an existing network attached storage device taxonomy. A first-order model quantifies performance differences of network attached storage architectures. Evaluations of several file systems from academia and industry explore the strengths and weaknesses of each approach.

3.1 Distributed File System Terminology

3.1.1 Client-Server Model

The client-server model is a distributed system resource management scheme; servers maintain shared resources for clients [44][45]. In the traditional sense, servers and clients are general purpose computers with processors, memories, and storage devices. Server computers act as file managers and storage servers. File managers maintain information specific to the file system, keeping track of file ownerships, directories, and locations of file metadata and real data. Storage servers administer and manage file system data. Networks connect clients with servers.

Modern storage devices are also storage servers. Like server computers, storage devices manage storage, cache data, and schedule requests. These devices service data requests

using block addressable, channel protocols. Channel protocols deliver considerably greater performance than network protocols. Modern storage devices can replace storage server computers. As a result, storage devices deliver substantially better performance at lower costs.

3.1.2 Server State

Servers are either stateless or stateful. Stateless servers maintain no information regarding the history of client requests. Since servers maintain no state information, requests must contain all necessary information to describe server tasks. Stateless behavior simplifies recovery; clients only need to resend requests that do not successfully complete.

Stateful servers maintain information about previous client requests. With this knowledge, clients and servers effectively prefetch and cache data. State information is difficult to rebuild if lost by server failures. Servers may write state information to stable storage in order to minimize lost state resulting from failures.

3.1.3 Caching Schemes and Consistency

Distributed file systems cache data at a variety of locations. These locations include main memories and storage devices of clients and servers. Data location largely determines client access times. Access times are shortest for data cached in client memories. Access times are longest when servers cache data on local storage devices. Access times of data stored on client devices and in server memories depend upon network delays and storage device performance.

Cache inconsistencies occur when clients modify one or more copies of the same data. To prevent or manage inconsistencies, distributed file systems have mechanisms to ensure coherence. These consistency mechanisms either invalidate or update stale copies of data. Stateless servers typically rely on clients to maintain consistency. Before accessing locally cached data, clients must verify consistency with servers. With stateless servers, clients perform write-through caching so that other clients receive modified data. Stateful servers take an active approach to consistency management. Servers perform call-backs to notify clients and other servers that cached data is inconsistent and must be written-back or invalidated.

Traditional client-server file systems only transfer real data between servers and clients, thereby maintaining all metadata on the servers. However, newer file systems migrate both metadata and real data to clients. Coherence of metadata is extremely important because inconsistencies affect file system integrity. Inconsistencies of real data may have negative consequences for users but do not affect the rest of the file system. Perfect consistency guarantees that data read by an application receives the most recently written copy. Although desirable, not all file systems provide perfect consistency.

3.1.4 Simultaneous File Sharing

File systems support various degrees of file sharing. The simplest is exclusive access. In this case, only one user may open a file for either reading or writing. Since files opened for reading are never modified, multiple readers may simultaneously access files without coherence problems. The file sharing state with an exclusive writer or any number of readers is known as non-write-shared. When a writer opens a non-write-shared file, the file state changes to write-shared.

File system consistency mechanisms support access states differently. Some file systems do not allow write-shared states, while others allow write-shared states without guaranteeing perfect data consistency. To compensate for a lack of perfect consistency, applications must use facilities, like advisor locks, to maintain real data consistency.

3.2 Network Attached Storage

Network attached storage (NAS) is a term with no precise definition. Marketing executives broadly use the NAS term to describe systems that include individual disk drives, robotic tape libraries, and dedicated server computers. To clarify terminology, Gibson *et al.* present a taxonomy of network attached storage architectures with regard to device roles in distributed file systems [9]. Four cases categorize devices:

Case 0 Server-Attached Disks (SAD)

Case 1 Server Integrated Disks (SID)

Case 2 Network SCSI (NetSCSI)

Case 3 Network-Attached Secure Disks (NASD)

3.2.1 NAS Architecture Taxonomy Summary

The taxonomy base case is server-attached disks (SAD). Disks locally attach to general purpose server computers. Servers transfer data between server storage devices and client memories. These transfers take place over server buses and network links. Protocols between servers and clients are usually different than protocols between servers and disks. Server-to-client protocols are typically high-level network protocols like TCP/IP. Transfers between servers and disks use channel protocols like SCSI.

Case 1 devices, known as Server Integrated Disks (SID), are specialized systems that only perform distributed file system server functions. These systems share the same basic architecture as SADs; client requests flow through servers to disks. Specialized systems typically use commodity disks with channel protocols, such as SCSI or IDE, and communicate with clients using high-level network protocols. The Network File System (NFS), developed by Sun, is a popular target for specialized servers [46].

Gibson defines case 2 as Network SCSI (NetSCSI). NetSCSI directly transfers data between clients and storage devices via a SCSI 3 network. Like SCSI 2 devices, NetSCSI devices are block addressable. File manager computers often facilitate file system operations and name-space manipulations.

Gibson proposes the case 3 approach called Network-attached Secure Disks (NASD). NASD is an enhanced device interface that supports higher level operations than NetSCSI. Unlike block addressable NetSCSI devices, the NASD interface is object addressable; NASD objects may be data extents or entire files. The NASD interface is generic enough to support multiple file system types. File managers operate on name-spaces by performing such functions as pathname traversals. NASD devices authenticate client requests with file managers to ensure security.

The four case Gibson taxonomy distinguishes the finer points of network attached storage. To generalize, case 2 should include interfaces other than SCSI. Examples of this expanded case are IPI-3 on HiPPI and, to some degree, the VMS System Communication Architecture (SCA), developed by Digital Equipment Corporation.

For the rest of this dissertation, an NAS prefix and case number suffix designate storage device architectures. For instance, NAS-0 refers to server-attached disks, while NAS-2 specifies the expanded definition of NetSCSI.

The traditional architecture classification is message-based and shared storage [47] [48]. Message-based approaches, also called shared nothing, fit under either NAS-0 or NAS-1, since computers store data locally and share data with other machines using a high-level, message passing protocol. Computers in a shared storage, or shared disk, configuration physically share storage devices on single channels. High availability systems² are examples of shared disk architectures. Multiple computers access shared devices; a high-level protocol maintains consistency. New interfaces like SCSI-3 combine message-based and shared storage features. For this reason, the NAS taxonomy provides a more precise description than message-based and shared storage classifications.

3.2.2 Analytic Model of Network Attached Storage

This section presents an analytic model of network attached storage architectures. This model provides a first-order calculation that quantifies regimes where an NAS-2 storage architecture may prevail over NAS-0 and NAS-1. Table 3.1 provides definitions of all variables of this section. This section combines the NAS-0 and NAS-1 cases into NAS-01, since the same analysis applies to both cases.

File systems often break user requests into smaller requests. For simplicity, this analysis models file system requests instead of user requests. The summation of file system request times determines user request times. Furthermore, network and channel device drivers may divide file system requests into smaller segments. Given n segments of length S , file system request sizes are nS in length.

Equations 3.1, 3.2, and 3.3 are linear equations that model segment transfer times (T). Each equation is a linear function of the segment size (S). Equation 3.1 is the disk transfer equation. Equations 3.2 and 3.3 are similar equations for networks and memories, respectively. These equations serve as first-order approximations.

²Many vendors sell high availability systems. These systems have backup servers sharing storage devices with main servers. If a main server fails, the backup server takes over server responsibilities.

Variable	
T	Segment request time
T_s	Start-up time
T_d	Data transfer time
R	Transfer rate
S	Transfer size
n	Number of request segments of size S
disk	Storage device (disk, disk array)
net	Network (physical and protocol)
mem	Memory system
s mem	Server memory system
c mem	Client memory system
T_1	Time to transfer data from disks to server network buffers
T_2	Time to transfer data from server to client user memory
T_3	Time to transfer data from disks to client user memory
T_{NAS01_S}	NAS-0 & NAS-1 serial time
T_{NAS01_P}	NAS-0 & NAS-1 pipeline time
T_{NAS2}	NAS-2

Table 3.1: **Model Variable Definitions** Variables used in all equations, figures, and plots of this section.

$$\begin{aligned}
T(\text{disk}) &= T_s(\text{disk}) + T_d(\text{disk}) \\
&= T_s(\text{disk}) + S/R(\text{disk})
\end{aligned} \tag{3.1}$$

$$\begin{aligned}
T(\text{net}) &= T_s(\text{net}) + T_d(\text{net}) \\
&= T_s(\text{net}) + S/R(\text{net})
\end{aligned} \tag{3.2}$$

$$\begin{aligned}
T(\text{mem}) &= T_s(\text{mem}) + T_d(\text{mem}) \\
&= T_s(\text{mem}) + S/R(\text{mem})
\end{aligned} \tag{3.3}$$

Figures 3.1 and 3.2 illustrate the data path between components within NAS-01 and NAS-2 systems, respectively. Data paths are divided into stages as illustrated in the figures. To simplify analysis, times T_1 , T_2 , and T_3 represent times through stages 1, 2, and 3. Equations 3.4, 3.5, and 3.6 represent these times.

$$T_1 = T(\text{disk}) + T(\text{s mem}) \quad (3.4)$$

$$T_2 = T(\text{net}) + 2T(\text{c mem}) \quad (3.5)$$

$$T_3 = T(\text{disk}) + T(\text{c mem}) \quad (3.6)$$

Figure 3.1 illustrates the NAS-01 data path between storage devices and user memory. Stage 1 is the data path between storage devices and server network buffers. This stage includes the server buffer cache as temporary storage for protocol translations. Some systems may combine the buffer cache and network buffer in order to eliminate data copies between temporary memory buffers. Proper data alignment is essential to accomplish this optimization. Furthermore, the network protocol must support such an optimization. Protocols like TCP calculate parity at the processors. Combined buffers may not support these protocols, since the buffer cache has no space for parity data.

Stage 2 transfers data between server network buffers and user space. This path includes a network transfer between server network buffers and client network buffers. Data transfers occur between client network buffers and the client buffer cache. A second transfer takes place between the client buffer cache and user space. Given proper alignment, an operating system capable of such an optimization may eliminate one or both memory copies.

Figure 3.2 illustrates the data path within an NAS-2 system. Data transfers between storage devices and user memory pass through an intermediate client buffer cache. Given proper data alignment, optimizations may avoid the buffer cache. The NAS-2 data path, labeled stage 3, requires time T_3 to transfer data.

In NAS-01 systems, transfers between storage devices and client user memory may either perform serially or in parallel. Figure 3.3 represents the serial transfer time of data between storage devices and client user memory. This figure illustrates a file system request broken into five segments labeled *A* through *E*. All five segments pass through stage 1 before any segment proceeds to stage 2. Equation 3.7 is a linear equation expressing the total serial transfer time of one request divided into n segments.

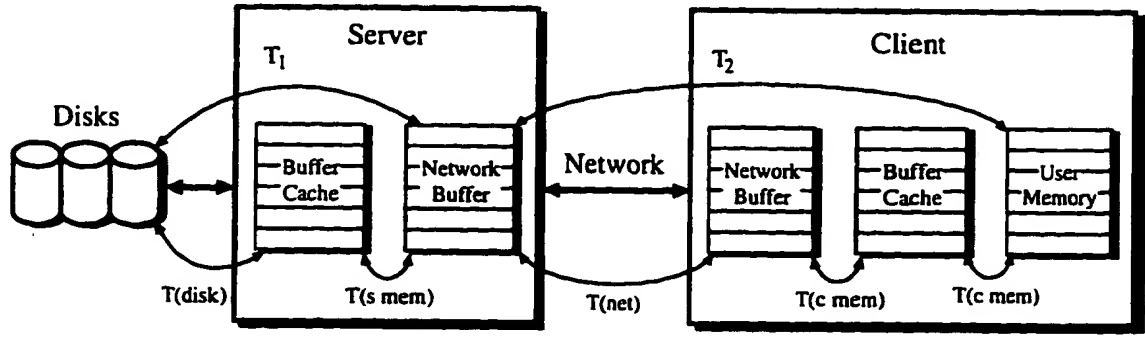


Figure 3.1: **Data Transfers within an NAS-01 System.** This figure illustrates data paths between storage devices and user memory in an NAS-01 system. Stage 1 is the path between storage devices and the server network buffer. Stage 2 is the path between the server network buffer and user memory.

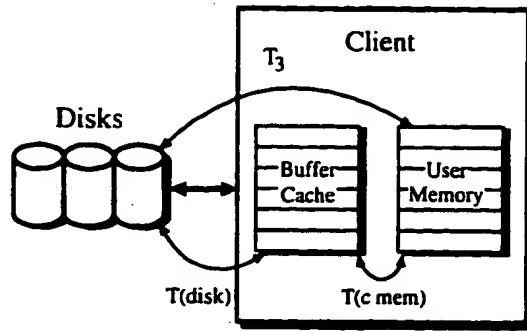


Figure 3.2: **Data Transfers within an NAS-2 System.** This figure illustrates the data path between storage devices and user memory in an NAS-2 system. Stage 3 is the path between storage devices and user memory.

Parallel transfers are more complex than serial transfers, since transport systems must break requests into segments in order to pipeline these segments through network and memory systems. Figure 3.4 represents a pipeline transfer; segments completing stage 1 continue to stage 2 without waiting for all segments to complete the first stage. Equation 3.8 expresses total pipeline transfer time of one request broken into n segments.

$$T_{NAS01_S} = n(T_1 + T_2) \quad (3.7)$$

$$T_{NAS01_P} = \begin{cases} T_1 + nT_2 & \text{for } T_2 > T_1 \\ nT_1 + T_2 & \text{for } T_1 > T_2 \end{cases} \quad (3.8)$$

$$T_{NAS2} = n(T_3) \quad (3.9)$$

Figure 3.5 represents request transfer times of NAS-2 systems. The five segments pass without interruption through stage 3 as one request. Equation 3.9 expresses the total transfer time of n segments.

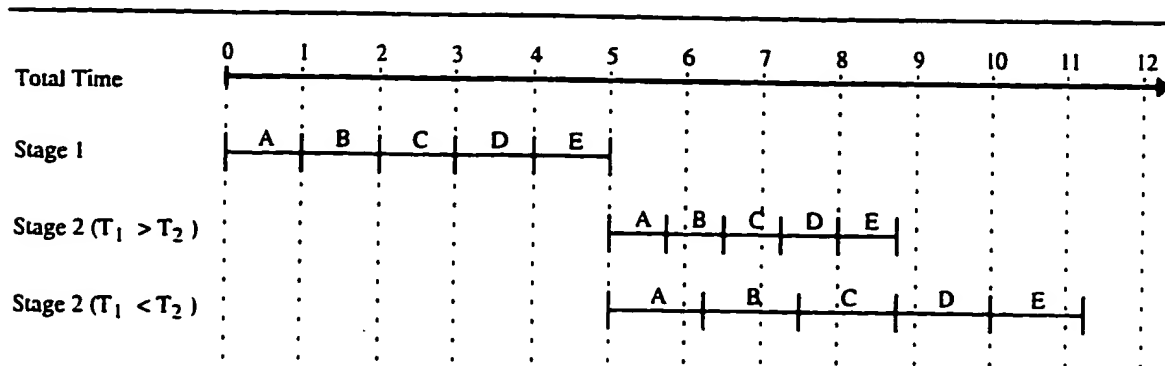


Figure 3.3: **NAS-01 Serial Data Transfer Time.** This figure illustrates the file system request transfer time of five segments labeled A through E. The transfer is serial, since all segments must complete stage 1 before starting stage 2. Segment E completes stage 1 at time 5; then, segment A begins stage 2.

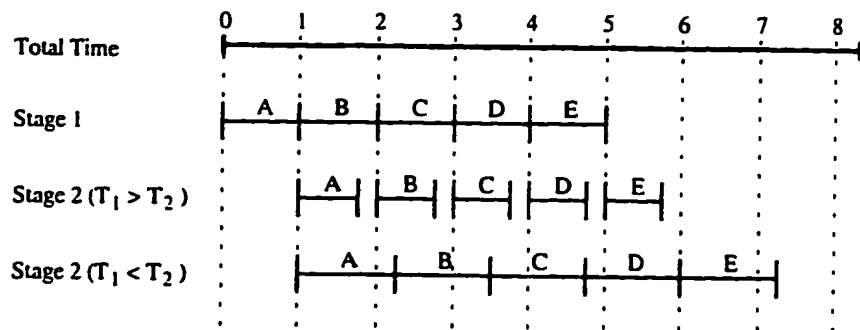


Figure 3.4: **NAS-01 Pipeline Data Transfer Time.** This figure illustrates the file system request transfer time of five segments labeled A through E. Data pipelines such that segments may begin stage 2 immediately after completing stage 1.

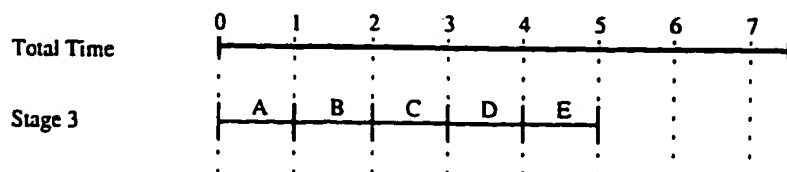


Figure 3.5: **NAS-2 Data Transfer Time.** This figure illustrates the file system request transfer time of five segments labeled A through E. The request completes at time 5.

3.2.3 Model Analysis

This section simulates two parameter sets for NAS-01 and NAS-2 architectures. The first set models a single disk with a startup time of 8 ms and a transfer rate of 10 MB/s. The second set models a disk array, or striped disks, with a startup time of 8 ms and a transfer rate of 75 MB/s. Both sets of parameters model high-speed networks with 1 ms startup times and 90 MB/s transfer rates. Model parameters for memory systems have no startup time and 500 MB/s transfer rates. Table 3.2 summarizes model parameters.

		Single Disk		Disk Array	
		NAS-01	NAS-2	NAS-01	NAS-2
Disk	Startup Time (ms)	8	8	8	8
	Transfer Rate (MB/s)	10	10	75	75
Server Memory	Startup Time (ms)	0		0	
	Transfer Rate (MB/s)	500		500	
Network	Startup Time (ms)	1		1	
	Transfer Rate (MB/s)	90		90	
Client Memory	Startup Time (ms)	0	0	0	0
	Transfer Rate (MB/s)	500	500	500	500

Table 3.2: **Model Parameters** This table lists model parameters for two sets of tests. The first set models a single disk and the second set models a disk array.

Figures 3.6 and 3.7 plot model performance for disk and disk array parameter sets, respectively. Each plot expresses the transfer rate performance for varying transfer sizes of NAS-01 serial, NAS-01 pipeline, and NAS-2 configurations. Each request size may represent multiple segment transfers between clients and storage devices.

Transfer times vary depending upon the number of segments. Plotted rates are from segment sizes that deliver the fastest request times. Results from NAS-01 serial and NAS-2 modeling are from tests where segment sizes equal request sizes. The fastest NAS-01 pipeline rates often come from tests with multiple segments. In reality, pipelining is difficult since the optimal number of segments per request varies depending upon system parameters and request sizes.

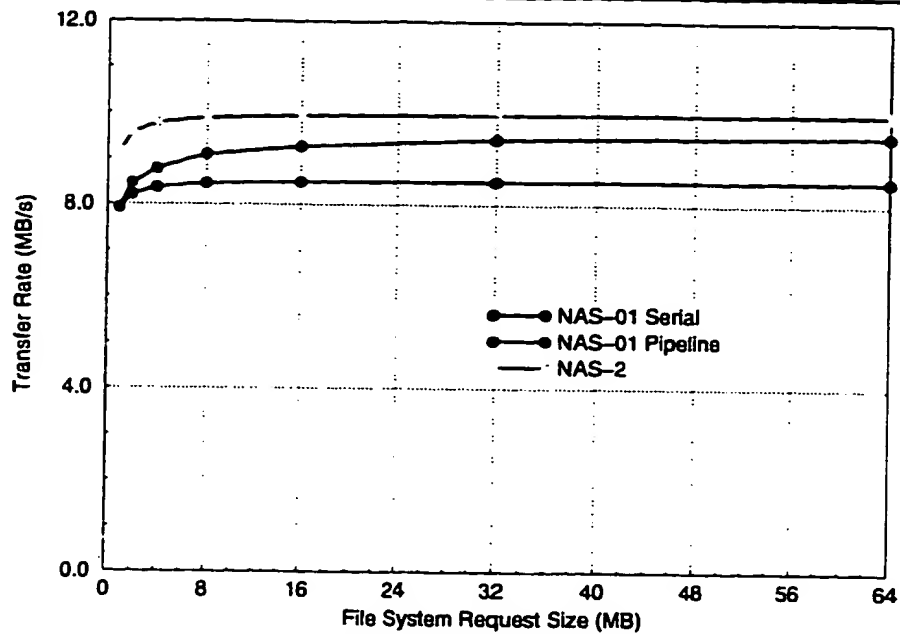


Figure 3.6: **Model Analysis of Single Disk Characteristics.** This figure plots model performance using single disk characteristics with slow transfer rates relative to network speeds.

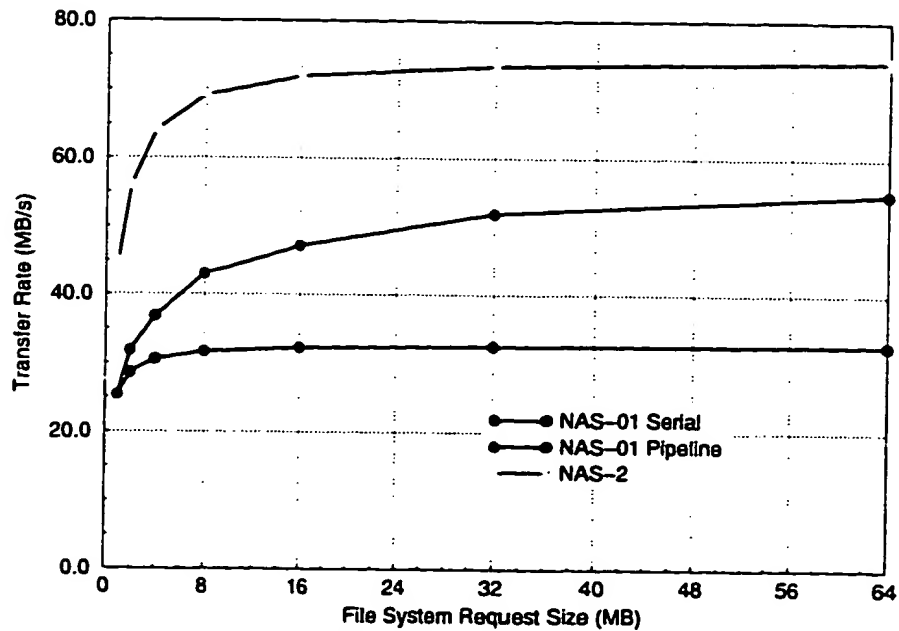


Figure 3.7: **Model Analysis of Disk Array Characteristics.** This figure plots model performance using disk array characteristics with transfer rates comparable to network speeds.

NAS-2 transfer rates are always higher than NAS-01. NAS-01 pipeline rates are similar to NAS-01 serial performance for small transfer sizes. NAS-01 pipeline rates approach NAS-2 performance for very large transfers. However, NAS-01 pipeline rates for the disk array parameter set do not exceed 65 MB/s until transfers larger than 256 GB.

With slow disk transfer rates relative to network speeds, NAS-2 demonstrates a minimal performance advantage over NAS-01. The second parameter set reveals the benefits of NAS-2 with respect to high performance storage devices. NAS-2 rates are almost twice that of NAS-01 serial. NAS-01 pipeline performance slowly approaches NAS-2 transfer rates. When device transfer rates are similar to network speeds, NAS-2 is the clear performance leader. Furthermore, this analysis does not model server workloads. Given overworked servers, NAS-2 may prevail at any device speed.

3.3 Distributed File System Architectures

The NAS taxonomy serves to characterize network attached storage device roles in distributed file systems. Alone, this taxonomy does not characterize distributed file systems, since the taxonomy does not delineate file manager designs. A complete characterization of distributed file system architectures must categorize file manager organization as well as classify network attached storage device usage.

File manager designs range from central server to merged client/server. Between central and merged designs is the distributed server architecture. Central servers manage all file system server functions. Distributed servers cooperate with other servers to satisfy client requests. Distributed server configurations often divide the file system name-space among servers. Each server is responsible for a portion of the name-space. Merged client/server systems place the majority of file management functions on clients. These clients service local requests and possibly requests from other clients. Some merged systems act as private file managers by only servicing local requests.

Some central server file system configurations resemble distributed server and merged client/server designs. For instance, several central server file systems could operate on different computers. Each server would provide different components of the file system name-space. Clients transparently access files within a single name-space from different servers. In effect, the total system would be a distributed environment. Central servers can also service local requests. In this case, the central servers resemble merged client/servers.

Central server designs may resemble distributed and merged configurations but lack redundancy and load balancing.

Figure 3.8 presents a diagram of distributed file system architectures [49] [50]. This diagram classifies distributed file systems based on file manager organization and network attached storage architecture. Shaded areas of the diagram illustrate existing file systems. The following sections describe these file systems.

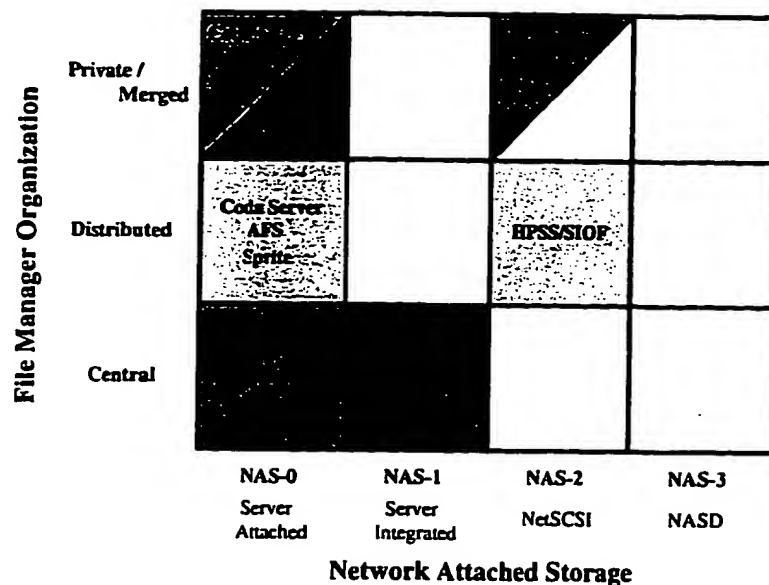


Figure 3.8: **Distributed File System Architectures.** This diagram classifies distributed file systems based on file manager organization and network attached storage architecture.

3.3.1 Network File System

The Network File System (NFS) was designed by Sun Microsystems in 1985 [1]. The goals of NFS include system independence, name transparency, and preservation of UNIX file system semantics. NFS is a central server, NAS-0 architecture. The server design is stateless, so clients make requests with all information necessary to complete operations.

Clients and servers communicate over a network using the remote procedure call (RPC) protocol. RPC is a high-level protocol built upon User Datagram Protocol (UDP) and Internet Protocol (IP). Transmission Control Protocol (TCP) may replace UDP to provide connection-oriented communication with guaranteed, in-order delivery. NFS fits into the OSI reference model at the application layer as shown in Table 3.3.

7	Application	NFS
6	Presentation	XDR
5	Session	RPC
4	Transport	UDP or TCP
3	Network	IP
2	Data Link	Interface Driver
1	Physical	Ethernet, ATM, Fibre Channel, etc.

Table 3.3: How NFS fits into the OSI Model

NFS servers are stateless and write modified data to stable storage before completing requests. The newest version, NFS 3, relaxes the requirement of writing to stable storage before request completion [51]. NFS maintains only a weak form of consistency, since single read and write requests may span several RPC operations. Multiple clients may issue overlapping requests.

Both NFS 2 and NFS 3 clients and servers cache data in system memory in order to improve read performance. Additionally, NFS 3 servers write cache. Clients also cache file attributes, but periodically invalidate these attributes to limit the use of stale data. Clients maintain file data consistency by verifying file modification times with servers.

The stateless server design is the crux of NFS simplicity. Servers use local file systems to store data; NFS does not manage storage. Primary server functions manage client requests and transport data. Server statelessness simplifies crash recovery. Failed clients do not affect the operations of servers or other clients. Servers that fail need only reboot; clients resend requests not completed within a given duration. Clients perceive failed servers as slow servers.

NFS offers portability and high connectivity but lacks fundamentals necessary for good performance. Single servers become bottlenecks as the number and size of client requests increase. NFS is distributed in the sense that multiple computers share files; however, the design is not distributed in a manner capable of providing scalable performance.

Single servers also make NFS vulnerable to failures. Some vendors provide high availability solutions. Backup servers take on the role of primary servers, if failures occur. These high availability solutions use shared storage techniques to connect backup servers to storage devices. Servers share devices, but device access is exclusive. RAID technologies improve storage device availability.

3.3.2 Sprite File System

Sprite is a distributed operating system for networked workstations. Sprite was developed at the University of California at Berkeley as part of the Symbolic Processing Using RISC (SPUR) research project [52]. This operating system distributes file server responsibilities across multiple servers. Each server stores data on locally attached disks. The primary goal of the file system is to provide name transparency while still providing adequate performance.

The Sprite file system maintains cache consistency using a server-initiated approach. Servers track open files. Clients cache non-write-shared files within local memories. When files move from non-write-shared to write-shared states, servers perform call-back operations and disable client caching.

The Sprite file system has two advantages over NFS. First, the file system distributes server workloads across multiple computers. Second, Sprite supports more aggressive client caching while maintaining perfect file consistency.

Srinivasan and Mogul modified a version of NFS to support the same consistency protocol as Sprite [53]. The modified NFS, called Spritely NFS, is a stateful design which benefits from improved caching. Performance enhancements are due primarily to the delayed write-back policy.

3.3.3 Andrew and Coda File Systems

The Andrew File System (AFS) was developed from a joint research project between IBM and Carnegie Mellon University [2]. Coda descended from AFS research [54]. Both AFS and Coda are designed to operate on distributed networks of workstations scaling up to 5000 machines. AFS and Coda use locally attached storage devices on both servers and clients.

AFS distributes the file system across multiple server computers. Like Sprite, AFS servers maintain state. Servers perform call-backs when client cached data is modified by other clients. Unlike Sprite, AFS only guarantees consistency at the granularity of the entire file. When multiple copies of a file exist, servers save the last file written.

Transarc Corporation took AFS technology and developed the Distributed File System (DFS). DFS is the basis of the Open Software Foundation (OSF) *Distributed Computing Environment* (DCE) [55]. DFS provides stronger UNIX consistency semantics than AFS.

Coda, which stands for “Constant Data Availability”, improves the availability of AFS. Clients cache entire files locally in memory and on disk. Furthermore, multiple file copies may exist on different servers. Single server failures have little impact on availability. Clients may also run in *disconnected operation mode*, thereby using only locally cached files. Disconnected clients later reconnect to the network and synchronize modified files with the distributed system.

Like AFS, Coda distributes file manager responsibilities to server computers, although Coda clients also perform server-like functions during disconnected operation. For this reason, the Coda client file manager organization is a merged client/server system with private file managers. However, Coda server design is that of a distributed organization.

3.3.4 Serverless Network File System

The Serverless Network File System (xFS)³ is part of the Network of Workstations (NOW) project at the University of California at Berkeley [56]. The NOW project continues Sprite research. xFS uses a log structured organization like the Log-structured File System (LFS) [57][58] and striping techniques from Zebra [59] to simplify failure recovery and provide high throughput transfers. Fast, switched networks connect xFS clients in an NAS-0 configuration.

The xFS project recognizes that central servers are performance and reliability bottlenecks. Therefore, xFS distributes traditional server responsibilities to the clients. Hence, any system can manage control directives, metadata, and real data. The serverless design attempts to improve load balancing, scalability, and availability.

Like Sprite, xFS supports aggressive client caching [60][61]. To maintain consistency, clients must acquire tokens for each file system block. Clients make non-local data requests to managers. If possible, these managers direct file requests to clients that have the data cached in memory; otherwise, the managers direct requests to appropriate storage servers.

³Berkeley xFS should not be confused with the SGI local file system known as XFS.

xFS differs from Sprite, Coda, and AFS, since xFS distributes metadata management across multiple nodes. In contrast, the other systems divide directory trees into subtrees and assign each subtree to different servers. xFS is a merged client/server architecture.

3.3.5 VAXcluster VMS File System

VAXclusters, developed by Digital Computer Corporation, are closely coupled structures of VAX computing and storage nodes [62][63]. Clusters operate as single systems. Each computing node runs the same versions of the distributed VAX/VMS operating system. Message-passing networks connect systems. The interconnection network has two topologies: the high-performance Star Coupler hub that supports a maximum of sixteen nodes and a low cost Ethernet network.

VAXcluster is a NAS-2 architecture; clients service only local file system requests. Storage devices connect to the system through Hierarchical Storage Controllers (HSC). The HSC provides a packetized, block-level interface. High-availability systems add a second HSC between dual ported disk devices and the network. This second HSC adds a redundant path between CPUs and storage devices. Each HSC supports up to thirty-two disks.

A distributed lock manager maintains shared file consistency. Systems request locks for particular access modes: exclusive access, protected read, concurrent read, and concurrent write. The lock manager queues resource requests until resources become available.

VMS allows caching of data and file system resources. The lock manager uses sequence numbers within each file synchronization lock. Upon modifying data blocks, VMS nodes increment sequence numbers in the file locks. Other nodes compare the lock sequence number to the cached data sequence number. Nodes refresh data from the HSC, if the sequence numbers are not equal.

3.3.6 SIOF/HPSS File Systems

The Scalable I/O Facility (SIOF) project at Lawrence Livermore National Laboratory has designed a network attached peripheral (NAP) interface to the High-Performance Storage System (HPSS) architecture [64][65]. HPSS is a distributed Hierarchical Storage Management (HSM) system based on the IEEE Mass Storage Reference Model version 5 [66].

SIOF NAP work supports TCP/IP network protocols. Future work will include channel interfaces like SCSI-3 on Fibre Channel.

The HPSS NAP architecture is a distributed server, NAS-2 architecture. The NAP work is just a portion of the architecture that includes interfaces like NFS and File Transfer Protocol (FTP). When completed, comparisons between HPSS file system options should reveal relative strengths of NAS-0 and NAS-2 architectures.

3.3.7 Cray Shared File System

The Cray Research Shared File System (SFS) is a distributed file system incorporated in the UNICOS operating system [67]. A HiPPI Maximum Strategies ND40/Gen 5 Storage Server serves data to each processing node. The Maximum Strategies device operates as a RAID 5 disk array with an IPI-3 command set. The disk array connects four C90 machines through a HiPPI switch. All C90 nodes privately service local requests; no server machine exists.

A Sun SPARC workstation connects to the HiPPI switch to perform arbitration for the HiPPI disk array. This workstation, call the HiPPI-SeMaPhore (HSMP), maintains mutual exclusion semaphores for data stored on the disk array. The HSMP performs error recovery functions, though this workstation is a single point of failure.

SFS supports two types of file operations: multiple readers and single writer. SFS provides consistency using semaphores to facilitate read-modify-write operations. Nodes are able to cache data like local file systems. However, the SFS consistency scheme limits open file states to non-write-shared.

3.3.8 IBM AIX Cluster File Systems

Devarakonda *et al.* evaluate two cluster file system designs [68]. The IBM Journaled AIX file system, JFS, is the basis of both designs. The first design is an NAS-0, central server architecture; the second design is an NAS-2, private client/server architecture. Calypso is a client-server distributed file system like NFS. The Calypso server uses local JFS to store file data. A token scheme maintains consistency between client caches and the server.

Parallel JFS (PJFS) uses a shared-disk approach like SFS and the VAXcluster. PJFS clients communicate with the logical volume manager (LVM) on a block interface server. The LVM

is similar to the VAXcluster Hierarchical Storage Controllers (HSC). PJFS only requires the LVM if devices are not sufficiently multi-ported. Since clients interface devices at a block level, PJFS is an NAS-2 configuration. Parallel JFS is a local file system modified to support multiple clients. PJFS maintains consistency by serializing accesses using a token mechanism. A token server manages these tokens. Like VAXclusters and Cray SFS, PJFS clients privately service local file requests.

Devarakonda *et al.* find that Calypso read throughput is as much as 30% higher than PJFS and two to four times better on write-shared files. PJFS delivers slightly higher performance than Calypso when files are locally cached. According to the study, a Calypso central server approach delivers higher performance than the PJFS shared-storage design. However, the authors also state that efficient metadata serialization is difficult if a file system is not originally designed for a cluster architecture. Local and cluster file systems typically require different optimizations.

3.4 Discussion

The current trend in distributed file system design deviates from centralized servers toward clients that perform server functionality. These clients may privately service local requests or work together to service an entire distributed system.

The analytical model, presented in this chapter, investigates network attached storage device performances. Storage devices deliver significantly higher throughput when configured as NAS-2 devices rather than NAS-0 or NAS-1 devices. Higher performance results from the direct path between storage devices and user memory. The model also reveals that devices much slower than network speeds may not benefit as much from NAS-2 architectures. In this case, the time taken to transfer data between servers and storage devices is considerably longer than the transfer duration between servers and clients. This analysis precludes server workloads; however, server loads would further decrease NAS-0 performance. In the realm of fast networks and slow devices, NAS-2 architectures still benefit from bypassing server machines.

Currently, Coda and xFS do not support NAS-2 storage devices. Coda, like AFS, targets large LANs and even WANS. xFS builds upon inexpensive workstations. NAS-2 storage devices may not be practical on large, non-secure networks. Network attached storage may not be an option for every machine in either of these environments. Standards, like

Fibre Channel, take time to reach mainstream, commodity markets. Until then, low-end workstations will use less-expensive networks and devices.

VAXcluster and the SIOF NAP interface have NAS-2 architectures. Both designs use high-level protocols to access storage, whereas Cray SFS uses the channel protocol, IPI-3. Network device drivers and modules of high-level interfaces often limit performance. Channel devices deliver higher performance.

The HSMP Sparc workstation is a single point of failure in SFS. SFS could provide a backup workstation to take over responsibilities should the primary HSMP fail. A better scheme would distribute lock management similar to the VAXcluster consistency scheme. However, obtaining good performance and high-availability with a distributed locking mechanism is a complex problem.

The three NAS-2 file systems, VAXcluster, PJFS, and SFS, are modified versions of local file systems. Although file system development benefits from code-reuse, such designs cannot optimize for multiple client operation. Caching, parallel metadata layout, and metadata serialization are topics that designers should address from the beginning of development.

Chapter 4 describes the Global File System (GFS). GFS is similar to SFS in that both file systems use readily available NAS-2 interfaces. Also, GFS clients privately service local file system requests. Unlike previously file systems, GFS uses a distributed lock manager controlled by storage devices. GFS clients perform distributed failure recovery. The file system prototype is written from scratch and optimized for an NAS-2 architecture. GFS design attempts to address high-performance needs for cluster environments. Chapter 4 compares and contrasts the Global File System with existing approaches.

Chapter 4

GFS Architecture and Implementation

We all want progress, but if you're on the wrong road, progress means doing an about-turn and walking back to the right road; in that case, the man who turns back soonest is the most progressive.

– C.S. Lewis

This chapter describes the architecture and implementation of the Global File System (GFS). The architecture sections introduce GFS design, discuss environments where the file system excels, and describe GFS advantages over traditional approaches. The implementation sections present the prototype design by describing file system structures, cache coherence mechanisms, and important file system routines.

4.1 The GFS Architecture

The Global File System is a distributed file system based on shared network storage. Client file managers exclusively service local file system requests. Network attached storage devices directly serve clients. Each client views storage as locally attached, though no single computer owns or controls these network attached devices. No direct communication exists between computers; GFS clients remain independent from failures and bottlenecks of other clients.

GFS achieves client independence by atomically modifying shared data. A storage-device-managed locking mechanism facilitates atomic operations. Before data modification, clients acquire locks. After clients modify and write data back to the storage devices, clients release the locks. In this regard, clients access storage devices like processors of a shared memory

multiprocessor computer (SMP) access memory. Except to maintain consistency, clients are unaware of other clients.

Storage devices connect to client computers through switched, channel networks called Storage Area Networks (SAN). GFS logically groups storage devices to provide clients with a unified storage space. This collection of network attached storage devices is a network storage pool (NSP). Subpools divide NSPs into groups of similar device types.

Figure 4.1 illustrates a GFS distributed environment. A cluster of independent clients connects to the network storage pool via a SAN. Each client may have multiple network connections to the NSP.

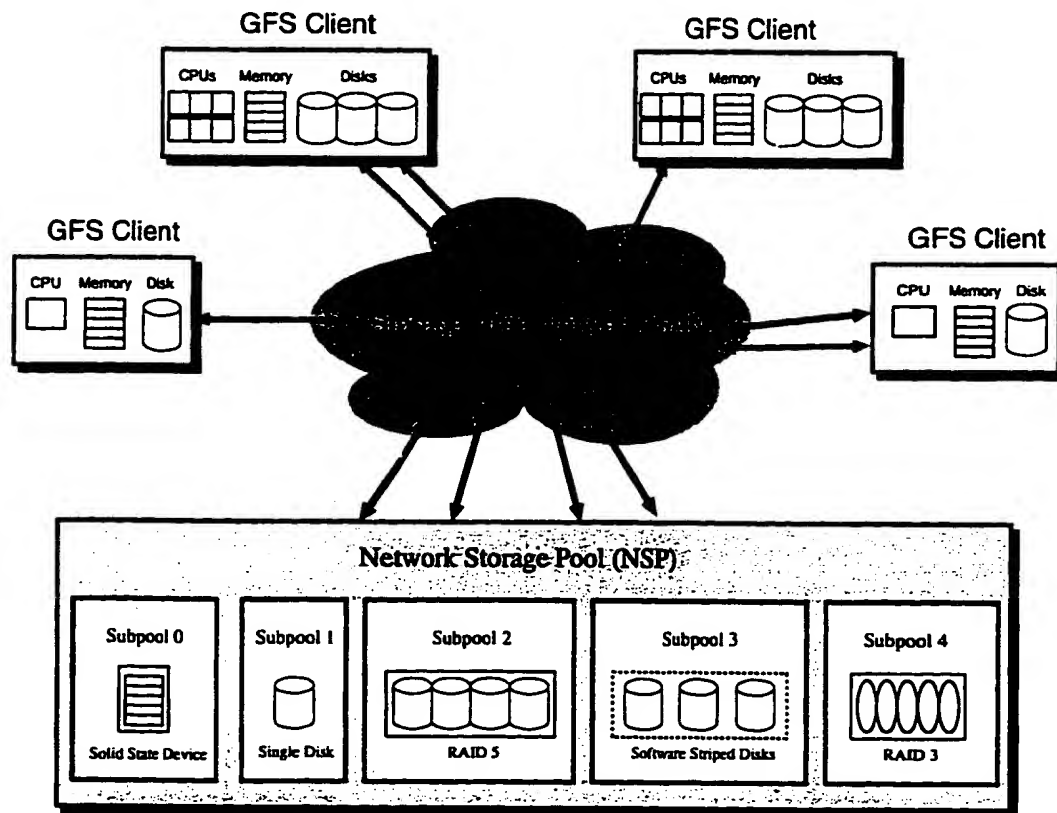


Figure 4.1: GFS Distributed System. This figure illustrates a GFS cluster. GFS clients access the network storage pool by means of a storage area network. Some clients have multiple network connections to the NSP.

4.1.1 Environment

GFS is a cluster-based distributed file system. Unlike distributed file system designs that focus on connectivity, GFS strives to deliver high-performance to a relatively small number of computers. GFS targets applications that require large storage capacities and bandwidth, such as multimedia, scientific computing, and visualization. Large storage capacities influence file system structures and caching policies.

Presently, the GFS prototype targets the SGI IRIX operating system; however, the architecture does not preclude a heterogeneous cluster. In fact, the prototype was developed on single processor, 32-bit desktop workstations; where as, this dissertation presents measurement results conducted on multi-processor, 64-bit machines. Cross-platform compatibility remains an implementation issue.

GFS does not provide security beyond the UNIX framework. Trusted environments are necessary to ensure UNIX protections. A secure environment is not possible without encryption and authentication schemes. Approaches like Network Attached Secure Disk (NASD) devices may improve GFS security, since NASD supports authentication schemes.

The GFS design focuses on achieving high-performance rather than scalable connectivity beyond a storage area network. A GFS client, however, can export file systems to computers not connected to the storage pool. In such situations, GFS clients act as servers for NFS or Hypertext Transfer Protocol (HTTP)⁴ clients [69]. However, export performance hinges upon architectures and configurations of both GFS and the export protocol.

Figure 4.2 illustrates an exported GFS configuration. A storage area network connects GFS clients to the network storage pool. A GFS client operates as an NFS server by exporting GFS to NFS clients. The NFS server and clients connect through a LAN. Another GFS client performs HTTP server functions by exporting data across the World Wide Web via WANs to HTTP clients.

4.1.2 Network Storage Pools

Network storage pools are collections of physically shared devices. Subpools partition NSPs according to device attributes. Subpools inherit characteristics from underlying devices and network connections. Device characteristics range from low-latency to high-

⁴HTTP is a stateless, application level communication protocol used by the World Wide Web.

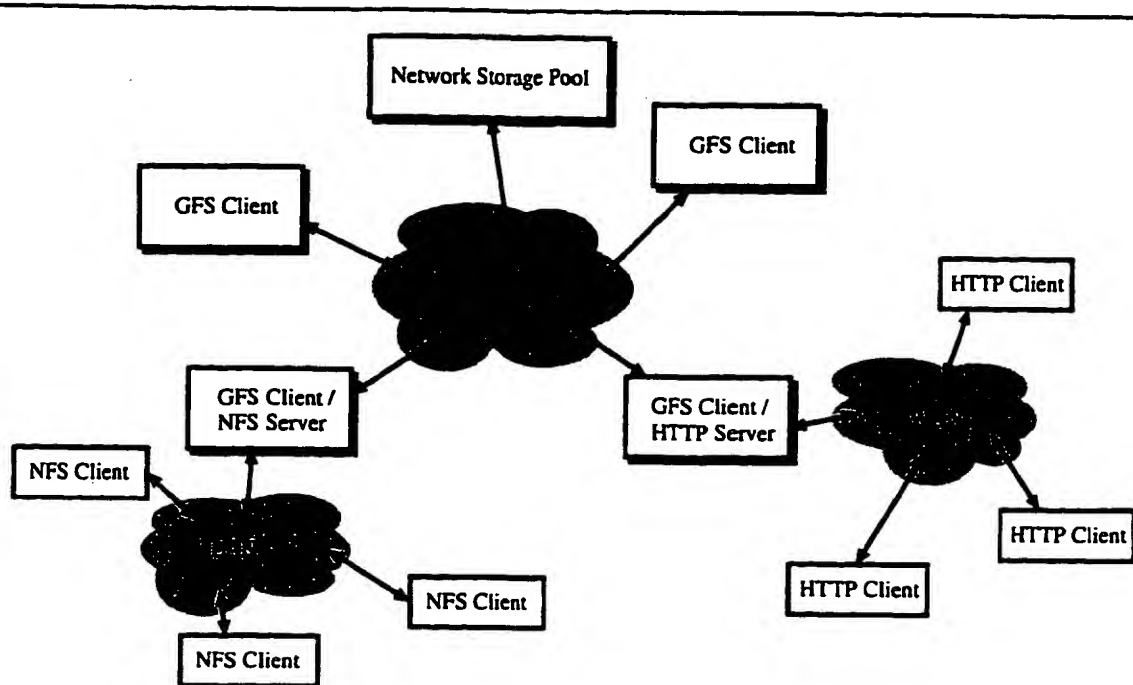


Figure 4.2: **Exporting GFS to NFS and HTTP.** This figure illustrates GFS clients exporting file systems to NFS clients and the World Wide Web. GFS clients serve NFS clients over LANs or HTTP clients across WANs.

bandwidth performance. A subpool of high-bandwidth devices inherits high-bandwidth characteristics. A high-bandwidth subpool includes disk arrays or software striped disks connecting to clients with one or more network links. A low-latency subpool consists of solid state devices. Robotic tape libraries constitute high-capacity subpools.

GFS exploits different subpool characteristics. GFS places frequently referenced files, like directories, on low-latency subpools. Large files benefit from high-bandwidth subpools. Single files may be split across subpools of different types. Although splitting real data in files is not useful, positioning real data and metadata on different subpools has performance advantages. The term for splitting real data and metadata is data-forking. GFS places real data on high-bandwidth subpools and metadata on low-latency subpools. Because metadata requires as little as one percent of the total storage space, data-forking conserves expensive low-latency storage

4.1.3 Memory Hierarchy

The Global File System memory hierarchy includes client memory, storage device caches, and storage device media. The GFS memory hierarchy is similar to that of a local file system; however, maintaining consistency between GFS levels is more complex. GFS uses a mutual exclusion locking scheme with a write-through approach for coherence.

Access times of each memory level differ by one order of magnitude. Buffer cache access durations include buffer cache searches and data access times. Buffer cache access times approach one hundred microseconds. Device cache access times incur device driver, network, and device overheads. Device cache access times range between one and three milliseconds. Accessing device media takes between three and twenty milliseconds. Media access times are roughly one order of magnitude longer than device cache access times and two orders longer than buffer cache access times.

Traditional local file systems underutilize storage device caches. Local file systems benefit more from main memory caching than from device caching. Because local file systems duplicate caching within system memory, disk cache reads of recently accessed data rarely occur. Device caches deliver a useful, intermediate cache level in the GFS memory hierarchy. This level bridges the latency gap between memory and magnetic media. Device caches provide a cache level shared by all clients. Clients benefit from this shared level, since data requested by one computer will likely be accessed by others.

Device caches typically use an LRU replacement policy. In addition to the LRU policy, GFS explicitly specifies data to be cached. GFS clients identify data that may benefit from caching, such as metadata and small files. Clients tag cacheable requests and send the commands to devices. Explicit caching performs better than the blind LRU policy of device caches, since non-cacheable data does not replace useful data. Directive caching is the GFS term for explicit specification of data caching.

Data consistency within and between all three memory levels of the GFS hierarchy is essential. Three different consistency mechanisms maintain coherence. Memory locks ensure mutual exclusion among processes in system memory. Storage devices manage consistency between local caches and media. GFS uses a locking mechanism, called device locks, to maintain consistency between system memories and devices.

Storage device controllers manage device locks. Clients must acquire device locks before modifying shared data. After writing modified data back to device storage, clients release the locks. Device locks not only ensure mutual exclusion but also provide a facility for

client memory caching. This locking scheme has the simplicity of a centralized mechanism yet distributes across several devices.

GFS mechanisms guarantee strong consistency; every device read returns the most recently written data. File system metadata requires strong consistency, since file system integrity cannot tolerate metadata inconsistency. To ensure perfect file consistency, GFS applies this strong consistency scheme to file real data. Perfect consistency guarantees that file reads return the most recently written data, even when files are write-shared by more than one client. Perfect file consistency is stronger than consistency found in most distributed file systems.

4.1.4 GFS Advantages

GFS has many advantages over existing distributed file systems:

- The symmetric multi-host design of GFS allows clusters of systems to behave as symmetric multiprocessor computers (SMP). GFS private file managers service only local file requests. Client resources, like memory, CPUs, and bus bandwidth, are not spent servicing requests for other clients. Furthermore, client failures have little, if any, direct effect on other computers.
- GFS takes advantage of an NAS-2 architecture by removing server machines from I/O paths. Figures 4.3 and 4.4 illustrate the levels of software and hardware that NFS and GFS must traverse, respectively. Many network and operating system layers require memory copies. In contrast to NFS, GFS provides direct control and data paths between storage devices and user memory.
- GFS was designed from scratch as a distributed file system rather than a modified local file system. Only ground-up development reasonably addresses caching, metadata serialization, and file system layout.
- GFS provides an architecture that logically groups storage devices into shared storage pools. GFS maps file system data to various storage subpools based on characteristics of underlying devices. Client machines directly realize device properties. Software and hardware redundancy mechanisms achieve data availability.
- GFS provides perfect file consistency. Read requests return the most recently written data, even for files shared by several users on different computers.

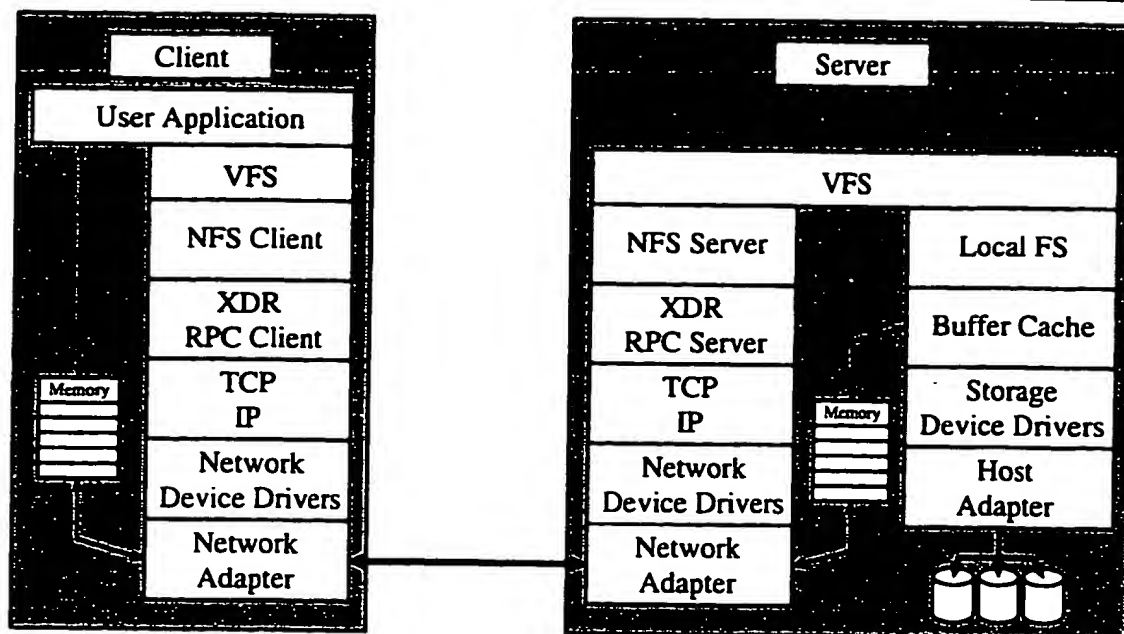


Figure 4.3: NFS Control and Data Paths. This figure illustrates NFS control and data paths. Control passes from user applications through VFS to the NFS client. NFS performs remote procedure calls over a TCP/IP network. The NFS server responds by accessing a local file system through VFS. This local file system makes requests to local storage devices. The data path consists of memory copies between user memory and the client buffer cache, the client buffer cache and network buffers, and network buffers and the server buffer cache. These copies transfer over system buses and network links.

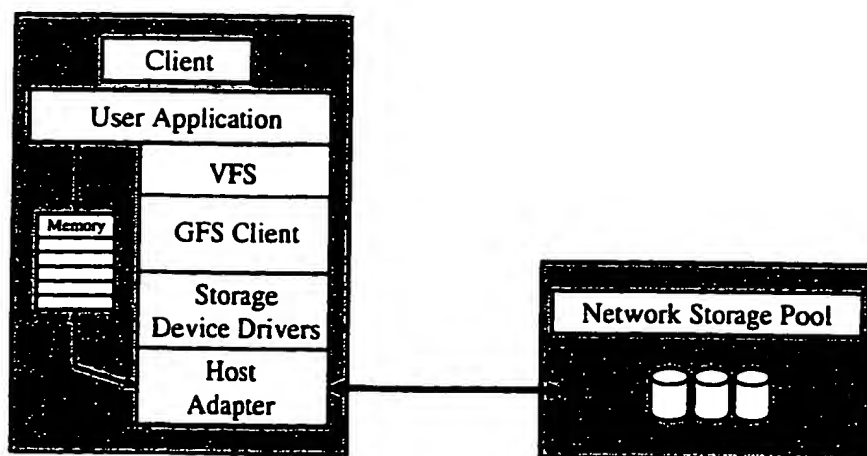


Figure 4.4: GFS Control and Data Paths. This figure illustrates GFS control and data paths. Control passes from client user applications through VFS to GFS. GFS makes requests to storage devices in the network storage pool. Block aligned data flows between user memory and storage devices. Non-aligned data temporarily passes through system memory.

4.2 File System Structure

GFS structures and algorithms differ considerably from traditional file systems. Unlike local file systems, GFS distributes file system resources across the entire storage subsystem. This distribution allows simultaneous access from multiple clients. GFS also attempts to place specific data types, either metadata or real data, on subpools with suitable performance. To achieve this placement, GFS must discern device types during file allocation. This section introduces file system layout and data placement. The following sections describe GFS coherence mechanisms and important file system routines.

4.2.1 Network Storage Pool Design

The network storage pool (NSP) provides each machine with a unified storage address space. A device driver layered on top of SCSI and Fibre Channel drivers implements the NSP. The NSP driver translates from the logical address space of the file system to the address space of each device. Subpools divide NSPs into groups of similar device types. These subpools inherit characteristics from underlying devices and network connections. GFS places file data on specific subpools in order to exploit device characteristics.

The network storage pool also manages device locks. GFS sends lock and unlock commands along with a logical lock number to the NSP. The NSP translates logical lock numbers into the physical numbers of appropriate devices. Until the lock operation is successful, the NSP polls the device with lock commands. The NSP also handles lock failure recovery.

NSP polling retry delays are between 1 ms and 5 ms. These delays deliver a balance between response time and device activity. Clients hold locks across several I/O operations with times totaling tens of milliseconds. Lock delay times of less than 1 ms could produce an enormous number of requests, while delays larger than 5 ms might unsatisfactorily increase response time.

As an alternative to the constant-retry-rate policy, adaptive schemes either increase or decrease polling rates after each retry. Increasing the retry rate decreases access times but gradually increases network traffic. Decreasing the retry rate reduces network and device activity, though increases response time. File system administrators must tune policies and rates for specific workloads.

4.2.2 Resource Groups

GFS organizes file systems into several resource groups (RG). Resource groups distribute file system resources across the entire NSP; multiple resource groups can exist per device. Resource groups are essentially mini-file systems. Each group possesses an information block, data bitmaps, dinodes, and data blocks. Resource group blocks contain information similar to traditional superblocks. In normal operation, resource groups are transparent to users.

Resource groups are similar to allocation groups (AG) in the SGI XFS local file system [70]. Like resource groups, allocation groups exploit this parallelism and scalability. AGs allow multiple threads of a single computer to allocate and free data blocks; RGs allow multiple clients to do the same. RGs also facilitate file placement on storage subpools.

Storage pools may consist of multiple devices. Several resource groups per device may also exist. If file accesses are evenly distributed among devices, parallelism enhances overall system performance. Advanced users and special utilities exploit this parallelism by moving files between resource groups. File migration balances activity between devices.

File data and metadata may span multiple resource groups and subpools. Figure 4.5 illustrates how GFS maps files from a directory tree to resource groups. In this example, the root directory resides on RG 0, file 10 exists on RG 6, and file 16 spans RG 6 and RG 7.

4.2.3 Superblock

The GFS superblock contains information that cannot be distributed across resource groups. Shared information includes the number of clients mounted on the file system and bitmaps to calculate unique identifiers for each client. Each client possesses an in-core superblock to maintain non-shared information. Non-shared information includes mount permissions and mounted NSP device names. The superblock also contains the static resource group index (RGI). The RGI describes the location and attributes of each RG.

4.2.4 Dinodes

Each GFS dinode occupies an entire file system block. GFS does not place multiple dinodes per block like most file systems, because single block sharing is not efficient in distributed

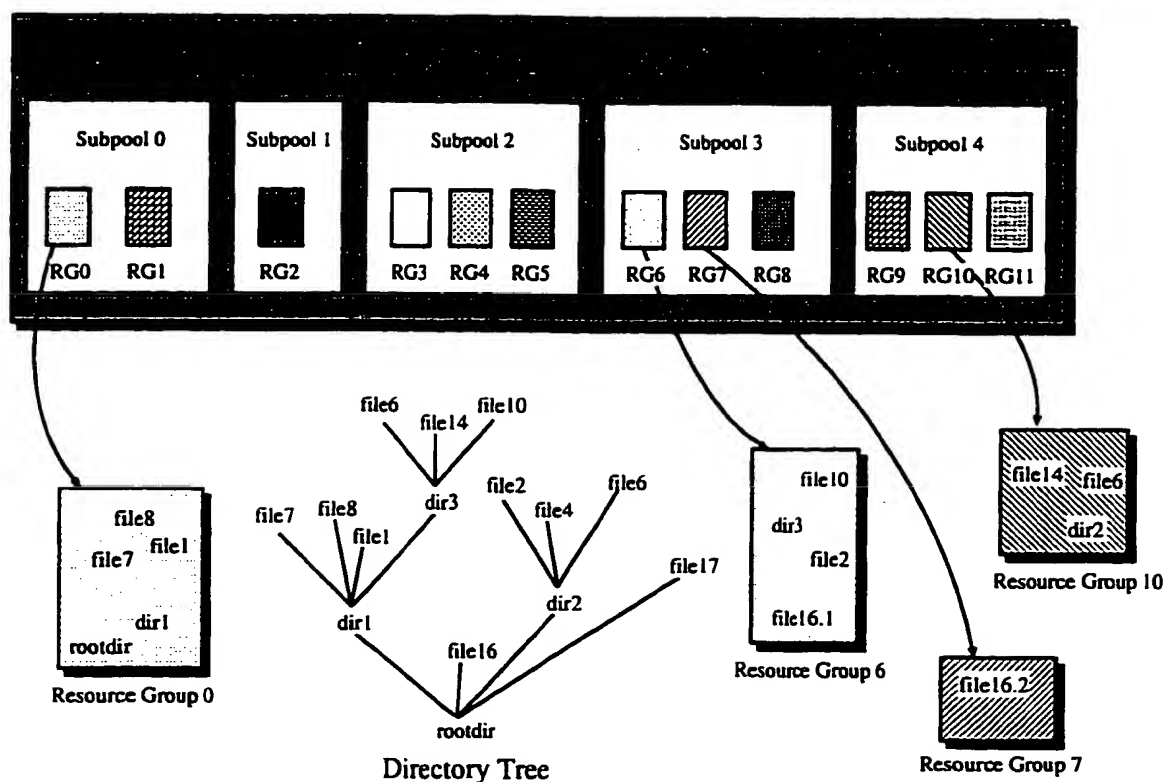


Figure 4.5: Files Mapped onto Resource Groups and Subpools. This figure illustrates the mapping of files onto resource groups and the mapping of resource groups onto NSP subpools. Files potentially span several resource groups and multiple subpools.

systems. As a result, the GFS dinode scheme incurs internal fragmentation, since dinodes rarely occupy entire blocks.

A GFS dinode consists of an information section and a data section. If the file size is larger than this data section, the dinode stores an array of pointers to data blocks or indirect data blocks. If the file fits into the data section, the space contains file data. Dinode stuffing is the technique that stores real data in a dinode. Stuffing compensates for internal fragmentation resulting from using entire file system block dinodes. Furthermore, clients completely transfer stuffed files with a single block request. Directory lookups often benefit from stuffing, since each pathname traversal requires one directory file read. Stuffed directory reads require less than half the requests of non-stuffed directory reads.

Consider a file system block size of 32 KB. Assume the dinode header requires 128 bytes. Without stuffing, a 1 byte file requires a total of 64 KB and at least two disk transfers to read the dinode and data block. With stuffing, a 1 byte file only requires 32 KB and one

read request. The file can grow to 32 KB minus 128 bytes, or 32,640 bytes, before GFS unstuffs the dinode.

GFS assigns inode numbers based on the disk address of each dinode. Directories contain file names and accompanying inode numbers. Once the GFS lookup operation matches a file name, GFS locates the dinode using the accompanying inode number. By assigning disk addresses to inode numbers, GFS dynamically allocates dinodes from the pool of free blocks.

GFS metadata trees differ from traditional UNIX file system (UFS) inode structures, illustrated in Figure 2.3. UFS inodes are tree structures with real data blocks at the leaves; the leaf may be differ in height. GFS uses the entire dinode data section as pointers. GFS trees have real data blocks at the leaves. All leaves have the same height. Figure 4.6 illustrates a GFS dinode and metadata tree with one level of indirection.

GFS dinode trees provide uniform access to real data. Given any file offset, GFS takes the same number of indirections through metadata to reach real data. Uniform tree heights deliver regularity to data access times. Furthermore, small file performance benefits from GFS dinode stuffing.

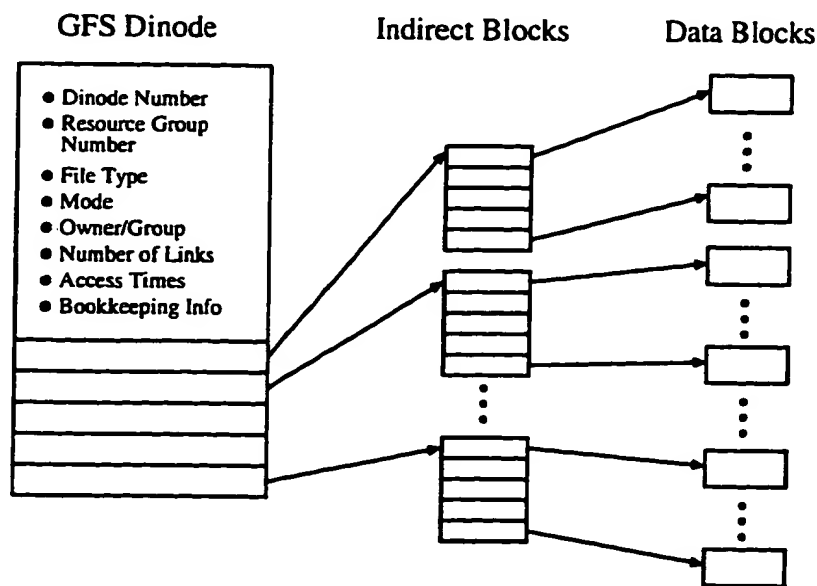


Figure 4.6: GFS Dinode and Metadata Tree. This figure illustrates a GFS dinode and metadata tree. The dinode has two sections: file information and data. Metadata trees are uniformly deep. In this example, the file system must traverse a single indirection in order to reach real data blocks.

4.3 Device Locks

The Global file system uses device locks to maintain data coherence. Device locks are mutual exclusion mechanisms managed by storage devices. GFS associates locks with file system data; devices have no knowledge of locked data. SCSI DLOCK commands manipulate device locks. In the prototype DLOCK command, each lock requires at most four bytes of device controller memory. Therefore, devices may provide thousands of locks with only minimal amounts of memory.

4.3.1 Lock State Bits, Activity Bits, and Clocks

Each device lock has a *state bit*, an *activity bit*, and a multi-bit *clock*. At completion of every DLOCK command, devices return the current values of each bit and clock. Clients save these values for lock activity measurements. Activity measurements are useful for failure recovery, load balancing shared resources, and maintaining data coherence.

Device locks support four primary actions: `lock`, `unlock`, `unlock increment` (`unlock incr`), and `reset lock`. These actions use test-and-set and clear operations to modify state bits. Devices increment clocks after successful completions of `unlock incr` and `reset lock` actions. `Reset lock` clears a lock, if the current clock value equals an input clock value.

Device locks support three secondary actions: `no action`, `activity on`, and `activity off`. These actions do not modify state bits. The `no action` command returns state bits, activity bits, and clocks. The `activity on` and `activity off` actions set and clear activity bits, respectively. Clocks increment after `activity off` actions. A set activity bit causes clocks to also increment after successful `unlock` actions. Table 4.1 summarizes the seven DLOCK operations.

Clock sizes are either 22 or 30 bits depending upon device configuration. Clients must be aware that clock values periodically roll-over from maximum value to zero. Clients determine roll-over by timing periods between each lock command. If current times differ from previous access times by less than the roll-over time, clocks have not rolled.

Action	Description
All Actions	To be set after ever action is performed Return.state \leftarrow Lock[N].state Return.activity \leftarrow Lock[N].activity Return.clock \leftarrow Lock[N].clock
No Action	No Operation Return.result \leftarrow 1
Lock	Test-and-Set Operation if (Lock[N].state = 1) Return.result \leftarrow 0 else Return.result \leftarrow 1 Lock[N].state \leftarrow 1
Unlock	Clear Operation Return.result \leftarrow 1 Lock[N].state \leftarrow 0 if (Lock[N].activity = 1) Lock[N].clock++
Unlock Incr	Clear Operation Return.result \leftarrow 1 Lock[N].state \leftarrow 0 Lock[N].clock++
Reset Lock	Conditional Clear Operation if (Lock[N].clock = <i>Input Clock Value</i>) Return.result \leftarrow 1 Lock[N].state \leftarrow 0 Lock[N].clock++ else Return.result \leftarrow 0
Activity On	Turn On Activity Monitor Lock[N].activity \leftarrow 1 Return.result \leftarrow 1
Activity Off	Turn Off Activity Monitor Lock[N].activity \leftarrow 0 Lock[N].clock++ Return.result \leftarrow 1

Table 4.1: Device Lock Operations

Roll-over time is measured by timing DLOCK commands. If DLOCK requires approximately 1 ms to lock and 1 ms to unlock, the shortest time for one clock increment is 2 ms. A 22 bit clock takes a minimum of $2^{22} \times 2 \text{ ms} = 2.33 \text{ hours}$ to roll. A 30 bit clock takes at least $2^{30} \times 2 \text{ ms} = 12 \text{ days}$ to roll. Given the long durations between roll-overs, roll-over detection is not difficult.

4.3.2 GFS Consistency and Device Locks

The Global File System uses device locks to maintain consistency between client memories and storage devices. GFS assigns locks to file system resources like resource groups and dinodes. Before reading or writing shared data, clients acquire appropriate device locks. When finished with this data, clients release the locks. Clients issue `unlock` commands after operations that do not modify data. Otherwise, clients use `unlock incr` commands to signify that data has been modified.

Clients cache data in memory, though data consistency is unknown until after the next successful lock action. Clients determine data consistency from returned clock values. Data is consistent if the returned clock has the same value as the clock returned from the previous `unlock` or `unlock incr` action. Otherwise, data is stale. Modified data is always written through to devices, so clients refresh stale data by reading from storage devices. This criteria assumes no clock roll-over. Given roll-over, clients treat data associated with the locks as inconsistent.

Clients often do not have foresight into whether data will be modified during atomic operations. However, devices increment clocks during `unlock` operations, so clients need not commit to data modification until after modifying the data. For instance, a client allocating data for a file does not know which resource groups have free data blocks. The client acquires the device lock associated with a resource group and then reads the RG block. If the RG has no free data blocks, the client releases the lock using an `unlock` command. Otherwise, the client allocates data blocks, writes the RG block back to storage, and releases the lock using an `unlock incr` command.

Table 4.2 gives a sequence of accesses from clients A and B. The example orders each event based on the time given. All commands access the same lock and activity monitoring remains off. The *lock* column lists the current state of the device lock and clock value, respectively. The *action* columns list commands sent to the device by clients. The *clock*

columns list the clock values stored by the clients. Client stored clock values update after command completion but do not change while the other client performs a command. Clock values of ? represent rolled clocks or locks never accessed. The *consistent* columns state whether or not cached data is consistent with data on the devices.

The first two lock commands from each machine indicate that data is not consistent, because clock values are unknown. The lock at time 7 guarantees that data is consistent, since the new clock value equals the old value. An unlock incr at time 9 signifies modified data. The lock at time 10 makes no guarantee that data is consistent, because the new clock differs from the old clock. The next lock also assumes data is inconsistent on the basis that the clocks differ. The final lock denotes consistent data, since the clock has not changed since time 12.

4.3.3 Client Failures

Client failures may leave device locks in locked states indefinitely. Active clients cannot acquire locks owned by failed clients. Active clients identify held locks from measurements of clock activity. A client that observes no lock activity will turn on activity monitoring for this lock. The clock now increments for both unlock and unlock incr operations. If the client observes activity, the lock is not held by a failed client. The client turns off activity monitoring. However, if the clock remains unchanged for a given time period during activity monitoring, the client resets the lock. This time period must be short enough to recognize failures in a timely manner yet not so small as to inappropriately reset locks. Reasonable times may range from thirty seconds to two minutes. Workloads and activity influence this time parameter.

The reset action compares an input clock with the current clock value. If the clocks are not identical, the reset fails. This input clock test ensures that locks will only be reset by clients that can identify current clock values. Locks are only reset once, even though multiple clients may attempt to reset the same lock.

4.3.4 Device Failures

Devices store device locks in volatile memories. After power failures or resets, devices lose lock states. Upon rebooting, devices zero lock states and notify clients of the power failures. Upon receiving these power-failure unit attentions, clients reacquire any lost locks.

Time	Lock State,Clock	Client A			Client B		
		Action	Clock	Consistent	Action	Clock	Consistent
0	0 3		?			?	
1	1 3	Lock	3	No		?	
2	1 3	Read-Only	3			?	
3	0 3	Unlock	3			?	
4	1 3		3		Lock	3	No
5	1 3		3		Read-Only	3	
6	0 3		3		Unlock	3	
7	1 3		3		Lock	3	Yes
8	1 3		3		Read-Write	3	
9	0 4		3		Unlock Incr	4	
10	1 4	Lock	4	No		4	
11	1 4	Read-Write	4			4	
12	0 5	Unlock Incr	5			4	
13	1 5		5		Lock	5	No
14	1 5		5		Read-Only	5	
15	0 5		5		Unlock	5	
16	1 5	Lock	5	Yes		5	
17	1 5	Read-Only	5			5	
18	0 5	Unlock	5			5	

Table 4.2: Device Lock Example This table gives an example of two clients accessing the same lock. The *time* column displays a time ordering of events from 0 to 18. The *lock* field gives state bits and clock values. The *action* columns show client actions. The *clock* columns list the clock values stored by the clients. The *consistent* column indicates whether cached data associated with the lock is consistent after a lock action.

Clients reissue lock requests in order to reacquire lost locks. Clients reissuing requests may compete with clients making new lock requests. Once the lock is acquired, the reissuing client verifies that the clock returned by the lock command is zero. If the clock is non-zero, another client has acquired and released the lock. The reissuing client must invalidate all metadata modifications performed before the device failure. This recovery is similar to recovery after client failures. Chapter 6 details possible recovery schemes.

4.3.5 Deadlocks and Starvation

Deadlocks occur as a result of two or more clients holding locks while waiting for other locks to become available. Devices are unaware of which client owns each lock, so devices neither detect nor break deadlocks. GFS uses a deadlock avoidance scheme. Clients acquire locks in ascending order of lock numbers.

Starvation occurs when clients cannot acquire locks. Two situations cause starvation: clients hold locks for extended periods of time and slow clients lose out to fast clients. GFS uses a fairness algorithm that holds locks for less than a maximum allotted time. Clients do not hold locks beyond independent file operations. This algorithm also prevents other clients from perceiving that locks are held by failed clients.

Starvation also occurs when lock ownership switches between fast clients, but slow clients are unable to acquire the lock. Slow machines can increase lock retry rates to solve this problem. Clients that perceive lock activity, though cannot acquire the lock, can decrease the delay between lock retries. Eventually, these clients will acquire the lock, since the delay between retries becomes negligible. For fairness, future lock requests must revert back to the old retry rate.

Starvation is a symptom of poor balancing. File system tools and system administrators should attempt to balance lock activity across all locks and resources. Lock activity measurements reveal hot-spots that must be cooled through redistributing resources to unloaded locks.

4.3.6 Comparisons with Existing Semaphore and Lock Mechanisms

Chapter 3 introduces the Digital VAXcluster and Cray SFS. Both of these systems are NAS-2 architectures and maintain consistency of shared data using locks and semaphores. Although both consistency schemes differ from GFS device locks, several device lock features resemble aspects of each scheme.

The VAXcluster distributes a lock manager across all processing nodes. This lock manager supports lock and unlock commands with specific access modes, such as exclusive, protected read, concurrent read, and concurrent write. The lock manager queues incompatible requests until resources become available. The locking scheme also supports an interrupt mechanism that notifies lock owners when other processes are waiting to acquire held locks.

The VAXcluster lock manager considerably differs from device locks. The lock manager is distributed across nodes of the VAXcluster instead of across network attached storage devices. However, the VAXcluster lock manager uses a counting mechanism that is similar to the device lock clocks. VAX nodes can associate 16-byte information blocks with each resource lock. Processes may modify this *value block* during lock releases. Processes acquiring this lock may use the value block to determine if cached data is consistent. Like device locks, if the value of the lock does not agree with the value of the cached data, the process must reload the data from disk storage. In addition to coherence management, GFS clocks serve the purposes of activity measurement for recovery and load-balancing.

The Cray Shared File System (SFS) has separate coherence mechanisms for each of the two configurations. The first configuration uses a high-speed semaphore device called the SMP. The SMP is expensive and manages only 2048 semaphores. The second configuration uses a Sun SPARC workstation, call the HiPPI-SeMaPhore (HSMP). The HSMP connects to the SFS cluster by a HiPPI network. This workstation maintains mutual exclusion semaphores and performs error recovery. Nodes acquire semaphores using test and set operations. If the test operation fails, nodes wait and retry later. The HSMP stores a node identifier associated with each set semaphore. The HSMP is an inexpensive solution with respect to the SMP, although the response time for the HSMP is 1 millisecond as compared to 6 microseconds for the SMP.

SFS consistency uses a two tier approach to locking. SFS semaphores serialize accesses to locks stored as data on devices; the on-disk locks serialize file system resource requests. SFS nodes acquire semaphores associated with file system resources. Each semaphore may lock multiple resources. After a node acquires a semaphore, the node reads the data from disk. Each resource maintains a field that identifies if the resource is locked. If the resource is locked, the node releases the semaphore, waits, and later retries the procedure. If the resource has not been locked, the node sets the lock field with an identifier and releases the semaphore.

To increase parallelism, SFS distinguishes between resource lock types. Nodes that do not modify resources use read-data locks. Writers use write-data locks. Only one process may hold a write-data lock, but many readers may acquire a read-data lock. Writers must wait until all readers release the read-data lock before continuing. This type of locking restricts SFS from supporting write-shared operations.

Both device locks and SFS semaphores serialize computer accesses to data stored on devices. GFS assigns device locks to file system resources just as SFS assigns semaphores

to resources. In both cases, several resources may share single locks. However, GFS does not use a two tier locking scheme. GFS holds locks during file accesses. Although locks may be held longer for GFS, clients do not perform I/O requests to modify locks stored on devices. Furthermore, device locks allow GFS to support write-shared caching, since device lock clocks inform GFS clients of potential cache inconsistencies.

Although device locks resemble other serialization mechanisms, three combined traits distinguish device locks from all other approaches. (1) Device locks are device managed. The mechanism is relatively simple to implement on commodity devices and does not overuse device CPU and memory resources. Several devices provide GFS clients with a distributed locking scheme. (2) Device locks provide a coherency scheme for GFS. GFS clients may cache write-shared data without danger of inconsistency. (3) Clock activity measurements allow GFS clients to initiate lock recovery as well as to load-balance lock accesses. Clocks enable client-initiated recovery without client-to-client communication. Clock activity is also a measure of lock usage. Since GFS allocates several resources to each lock, clients may access some locks more frequently than other locks. Activity measurements quantify which resources may benefit from lock reassignment.

4.4 File System Vnode Operations

Table 4.3 lists the mapping between common system calls and vnode operations. The vnode operations call corresponding GFS vnode routines. Several important GFS functions are summarized in the list below.

<code>gfs_lookup</code>	Searches through the directory path in order to locate the given filename. Each iteration of lookup locks the directory dinode, reads the directory using <code>gfs_read</code> , searches for the filename, and then unlocks the dinode. If the file is found, <code>gfs_lookup</code> allocates a vnode and an inode for the file. Allocating an inode involves atomically incrementing the dinode open count. This atomic operation includes locking, reading, modifying, writing back, and unlocking the dinode.
<code>gfs_access</code>	Verifies that the user has permission to access the file.
<code>gfs_open</code>	Does nothing.

System Call	VNODE Operation	GFS Routines	Description
open	VOP_LOOKUP	gfs.lookup	Search directory path in order to locate filename.
	VOP_ACCESS	gfs.access	Verify access permissions.
	VOP_OPEN	gfs.open	Open file.
creat	VOP_LOOKUP	gfs.lookup	Search directory path in order to locate filename.
	VOP_ACCESS	gfs.access	Verify access permissions.
	VOP_CREATE	gfs.create	Create file.
	VOP_OPEN	gfs.open	Open file.
close	VOP_CLOSE	gfs.close	Close file and free inode.
	VOP_INACTIVE	gfs.inactive	Release vnode.
read	VOP_READ	gfs.read	Read from file.
write	VOP_WRITE	gfs.write	Write to file.
readdir	VOP_LOOKUP	gfs.lookup	Search directory path in order to locate filename.
	VOP_GETTATTR	gfs.gettattr	Check directory attributes.
	VOP_ACCESS	gfs.access	Verify access permissions.
	VOP_OPEN	gfs.open	Open directory file.
	VOP_READDIR	gfs.readdir	Read directory entries.
	VOP_CLOSE	gfs.close	Close directory file.
unlink	VOP_INACTIVE	gfs.inactive	Release directory vnode.
	VOP_LOOKUP	gfs.lookup	Search directory path in order to locate filename.
	VOP_REMOVE	gfs.access	Remove file.

Table 4.3: System Call Mapping to GFS Functions This table gives a mapping between common system calls and vnode operations. A one-to-one correspondence exists between vnode operation calls and GFS routines that implement the operations.

gfs.create Creates a new file if the file does not exist. `gfs.create` allocates a vnode, creates and initializes a dinode, and adds the filename to the directory. To create a dinode, GFS allocates an available data block. This allocation locks a resource group, reads the resource group block, and searches for a free data block. Once found, GFS sets the bitmap for the block, writes back the resource group block, and unlocks the resource group. `gfs.create` adds a filename to a directory by locking the directory dinode, reading the directory, adding the filename, writing the directory to disk, and unlocking the dinode.

<code>gfs_close</code>	Closes the file by freeing the inode. Freeing an inode involves atomically decrementing the dinode open count. If the dinode is marked for removal, GFS releases the file metadata and real data by clearing the appropriate resource group bitmaps.
<code>gfs_inactive</code>	Does nothing, since the file is already closed.
<code>gfs_bmap</code>	Maps the user read or write request to file system block address space. For writes, the routine might grow the dinode metadata tree by allocating and initializing pointer blocks. Once the metadata tree is complete, <code>gfs_bmap</code> allocates real data blocks. GFS allocates all blocks from resource groups by locking the resource group, reading the resource group block, setting bitmaps corresponding to free data blocks, writing back the resource group block, and then unlocking the resource group. Every block allocation step requires multiple disk reads and writes. To reduce this overhead, <code>gfs_bmap</code> traverses through the metadata tree only once. GFS performs all allocation in one step and then sequentially writes the newly created metadata to storage.
<code>gfs_read</code>	Transfers data from storage devices to user memory. <code>gfs_read</code> first locks and reads the dinode and then calls <code>gfs_bmap</code> . <code>gfs_bmap</code> returns a structure that describes disk read operations for <code>gfs_read</code> to perform. These reads directly transfer data into user memory, if block-aligned; otherwise, the reads transfer data into temporary system memory and then to user space. The function completes by unlocking the dinode.
<code>gfs_write</code>	Transfers data from user memory to storage devices. <code>gfs_write</code> first locks and reads the dinode and then calls <code>gfs_bmap</code> . <code>gfs_bmap</code> returns a structure that describes disk write operations for <code>gfs_write</code> to perform. These writes directly transfer data from user memory, if block-aligned; otherwise, <code>gfs_write</code> reads the target block into system memory, copies user data to this memory, and writes the data block back to disk. The function completes after setting the file modification time, writing the dinode back to disk, and unlocking the dinode.

<code>gfs_readdir</code>	Reads the contents of a directory and translates the GFS directory structure into a file system independent format. <code>gfs_readdir</code> locks the directory dinode, reads the contents using <code>gfs_read</code> , translates each directory entry into an operating system independent format, copies the entries to user space, and then unlocks the dinode.
<code>gfs_getattr</code>	Returns file attributes. <code>gfs_getattr</code> reads the dinode and copies file attributes to user space.
<code>gfs_setattr</code>	Modifies file attributes. <code>gfs_setattr</code> locks the dinode, reads the dinode, modifies file attributes, writes back the dinode, and unlocks the dinode. <code>gfs_setattr</code> truncates files by freeing metadata and real data blocks and zeroing dinode pointer sections.
<code>gfs_remove</code>	Removes a filename from a directory. If the file is currently open, <code>gfs_remove</code> marks the dinode for removal and then returns. If the file is not open, <code>gfs_remove</code> frees file metadata and real data by clearing the appropriate resource group bitmaps.

4.5 Discussion

This chapter describes the architecture and implementation of the Global File System. The architecture distributes server functionality to clients and network attached storage devices. GFS private file managers service local file system requests. The implementation details include file system layout, device lock consistency mechanism, and some important vnode operations.

GFS differs from existing file systems in several ways. These differences are improvements over existing techniques. NAS-0 and NAS-1 file systems, such NFS, Coda, and xFS, do not have the direct data path advantages of NAS-2 architectures. The analytical model from Chapter 3 demonstrates performance benefits of the NAS-2 direct data path.

GFS design and implementation differ from existing NAS-2 file systems. VAXclusters, SFS, and PJFS are modifications of local file systems. These systems serialize client accesses to shared data but cannot adequately parallelize the file system as a whole. GFS is designed from scratch to distribute file system resources across the entire storage pool. GFS design addresses caching, serialization, and file system layout.

Chapter 5

GFS Performance Analysis

Measurements are not to provide numbers but insight.

– Ingrid Bucher

This chapter investigates performance characteristics of the Global File System (GFS). GFS benchmarking focuses on sequential read and write accesses to entire files. This analysis compares GFS performance to raw performance of the storage subsystem. Analysis concentrates on file sizes between 1 MB and 256 MB. This range represents a realm where GFS provides a critical throughput advantage over existing systems.

This analysis does not investigate performance of file system utilities. Such utilities create and remove directories, list contents of directories, and obtain file statistics. Although utility response times are longer for GFS than local file systems, the duration takes the same order of time as other distributed file systems like NFS on 10 Mbit/s Ethernet.

This analysis quantifies GFS overhead beyond raw subsystem performance. GFS overheads include additional metadata accesses, processing activity, and consistency management. For comparison, random and sequential raw performance analysis includes single disks, striped disks, and disk arrays. Analysis contrasts single client performance with raw sequential performance in order to quantify GFS overheads. Single client performance is also the baseline for multiple client scaling measurements. This chapter presents multiple client aggregate performance and scaled speedup. Finally, the study investigates directory performance to quantify file lookup overheads.

5.1 Test Environment

The test environment consists of a homogeneous cluster of four Silicon Graphics Power Challenge shared memory, multi-processor computers. Tests include measurements from eight Seagate Barracuda 9 Fibre Channel disk drives and four Ciprico 7000 series RAID 3 disk arrays. Each Ciprico array contains 8 data and 1 parity disks connected on separate wide SCSI buses. A Brocade Silkorm FC switch connects Ciprico disk arrays with SGI Challenge computers. A Fibre Channel loop connects Seagate drives to a single Challenge machine. Currently, fabrics do not support Seagate disks.

This equipment represents a relatively high-end, multi-million dollar configuration. However, the GFS architecture does not preclude lower priced workstations and PCs. Performance of less powerful computers may not differ drastically from high-end counterparts, since I/O-intensive workloads tend to dominate processing power and memory bandwidths. Early tests reveal that low-end SGI workstations deliver approximately 10% less bandwidth than high-end counterparts.

The following list summarizes the test equipment. Figure 5.1 is a diagram of the test configuration.

Computers	Silicon Graphics Power Challenges with 194-MHz MIPS R10000 processors. All systems have between 512 MB and 2 GB of 4-way or 8-way interleaved memory.
Disk Arrays	Ciprico 7010 RAID 3 disk arrays containing 8 data and 1 parity Seagate Barracuda 9 (ST19171WD) disk drives. At 9 GB of storage per disk, the total capacity of each disk array is 72 GB. Each Seagate disk contains a 512 MB disk cache.
Disk Drives	Seagate Barracuda 9 (ST19171FC) Fibre Channel disk drives. Each drive has a 512 MB cache and a 9 GB magnetic storage capacity.
Host Adapters	Prisa HIO-FC single-ported and dual-ported host adapters.
Switches	Brocade Silkorm 16 Port Fibre Channel switch.

Whenever possible, measurements express transfer rates in units of megabytes per second (MB/s). Although disk manufacturers represent 1 MB as 10^6 bytes, this analysis defines

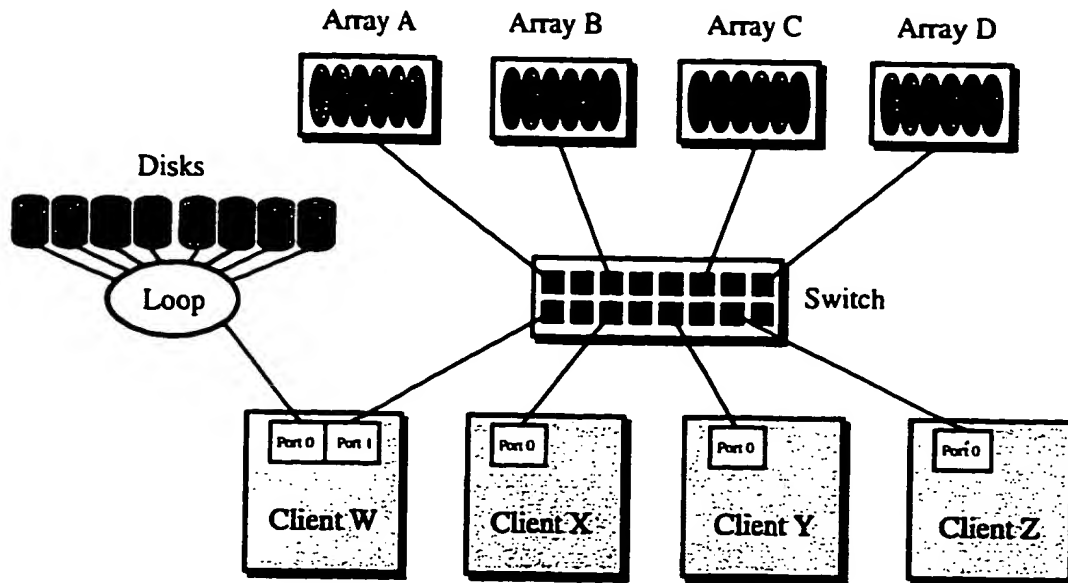


Figure 5.1: **GFS Test Configuration.** Each of the four Ciprico disk arrays connect to the Brocade Silksworm Fibre Channel switch. The switch then connects to four SGI Challenge machines labeled as clients W through Z. The FC loop connects Eight Seagate Barracuda 9 disk drives to one SGI Challenge.

1 MB to be 2^{20} or 1,048,576 bytes. Statistics derived from 10^6 byte conversions are five percent larger than 2^{20} byte conversions; hence, 10^6 conversions make better marketing statistics. However, these performance tests transfer data using multiples of two request sizes to ensure proper alignment of memory and disk accesses. For clarity and accuracy, this dissertation uses 2^{20} to convert between bytes and megabytes.

5.2 Storage Subsystem Performance Characteristics

This section analyzes storage subsystem performance characteristics of a single computer on a set of disks and disks arrays. Raw performance is the maximum performance that any file system may deliver without caching. For this reason, raw performance is the baseline for GFS comparisons. In addition to read and write performance, this section measures average device lock command times.

The following analysis presents random and sequential access characteristics as distinct evaluations. Random access performance is important, since many file system references, though definitely not random, cause storage devices to seek between tracks. Such accesses

also tend to vary in transfer length. Sequential access performance is as important, since many applications sequentially reference files. File systems allocate file data to exploit this reference behavior in order to minimize disk seeks.

Many system components affect performance of storage subsystems. The intent of this analysis is not to thoroughly characterize individual components but rather treat all components as one subsystem. This analysis investigates single disk, striped disks, and disk array performance. Storage devices are the only subsystem components to change during these tests.

5.2.1 Device Lock Performance

Device lock performance tests time lock and unlock commands. The lock operation takes approximately the same time whether successful or not. On average, Seagate disks requires 1.29 ms for locking and 1.05 ms for unlocking. Ciprico arrays take 1.31 ms for locking and 1.09 ms for unlocking.

Measurements represent dedicated device performance. The tests time each command with no system or device activity. Commands pass from user space through the NSP device driver using an `ioctl` call. The times include device driver, network, and device overheads. With normal GFS activity, lock commands take longer, since other requests contend for the same networks and devices.

5.2.2 Random Access Performance of Raw Devices

By nature, random access measurements are difficult to conduct and nearly impossible to repeat; therefore, this analysis focuses on trends instead of absolute measurements. This study conducts read and write tests by sending requests of random lengths and block addresses to storage devices. Tests uniformly distribute random length requests up to 16 MB and 32 MB for Seagate disks and Ciprico arrays, respectively. Requests to Ciprico arrays align accesses on 4 KB (eight 512-byte block) boundaries to eliminate read-modify-writes of parity disks. Tests also uniformly distribute starting block addresses between the first device block and the last device block that supports the maximum transfer length.

Each test times only `read` or `write` systems calls. Memory allocation and `open` and `close` system calls are not included. Before each read or write request, the test program

performs an `lseek` system call that sets the file pointer to a given offset. Devices do not seek during this operation. The test program determines this offset from a uniform distribution random number generator. The random number generator also determines the length of each request.

Figures 5.2 and 5.3 plot performance of one thousand read and write requests to a Seagate disk. Figures 5.4 and 5.5 plot one thousand read and write requests to a Ciprico array. Quantization of points into distinct lines results from accesses to different disk zones; outer zones have higher transfer rates than inner zones. The inverse of the slope of the lowest lines estimates the maximum device transfer rates. The y-intercepts approximate minimum device access times.

Seagate disks have maximum transfer rates of 10.4 MB/s for reads and writes. Minimum access times for reads and writes are approximately 15 ms and 3 ms, respectively. Disk caches buffer write data until devices seek to starting addresses. Without the write cache, write access times are approximately the same as read times.

Ciprico arrays have maximum transfer rates of 83 MB/s for reads and 78 MB/s for writes. Minimum access times for reads and writes are approximately 15 ms and 3 ms, respectively. Read requests complete when the slowest data disk finishes reading. Surprisingly, read access times of arrays are close to those of single disks.

Minimum read access times significantly depend upon disk seek times and rotational latencies. As a result, access times deviate from the mean by as much as 10 ms. This variance is more significant for small transfers than large. However, transfer times of large requests dominate seek and rotational latencies. Minimum write access times do not deviate as much as read access times. Write cache access times are nearly constant.

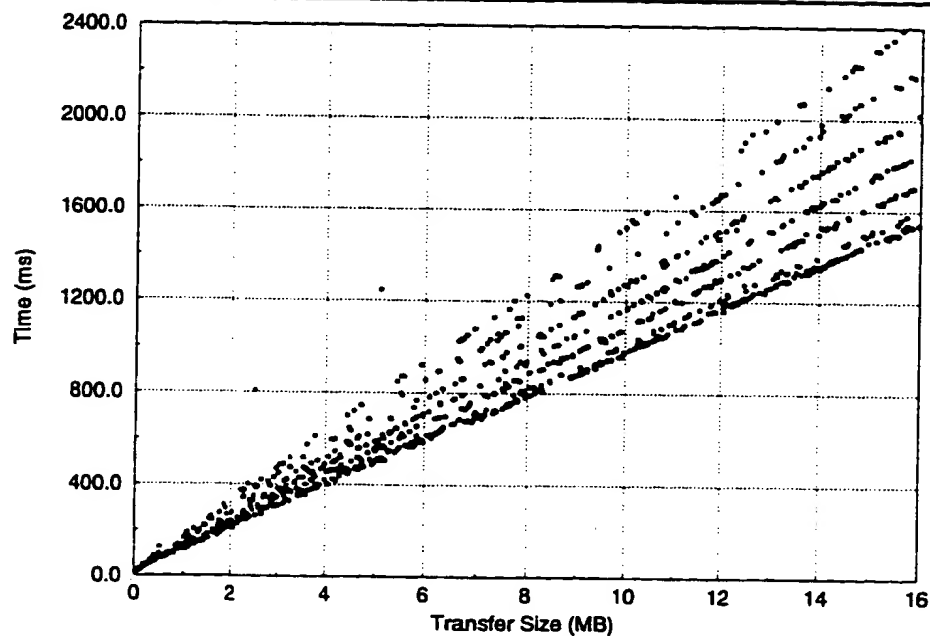


Figure 5.2: Seagate Barracuda 9 Random Access Read Performance. This figure plots read requests of random lengths from random locations on the disk. The plot shows a maximum transfer rate of 10.4 MB/s and a minimum access time of approximately 15 ms. Quantization of points into distinct lines results from accesses to different disk zones.

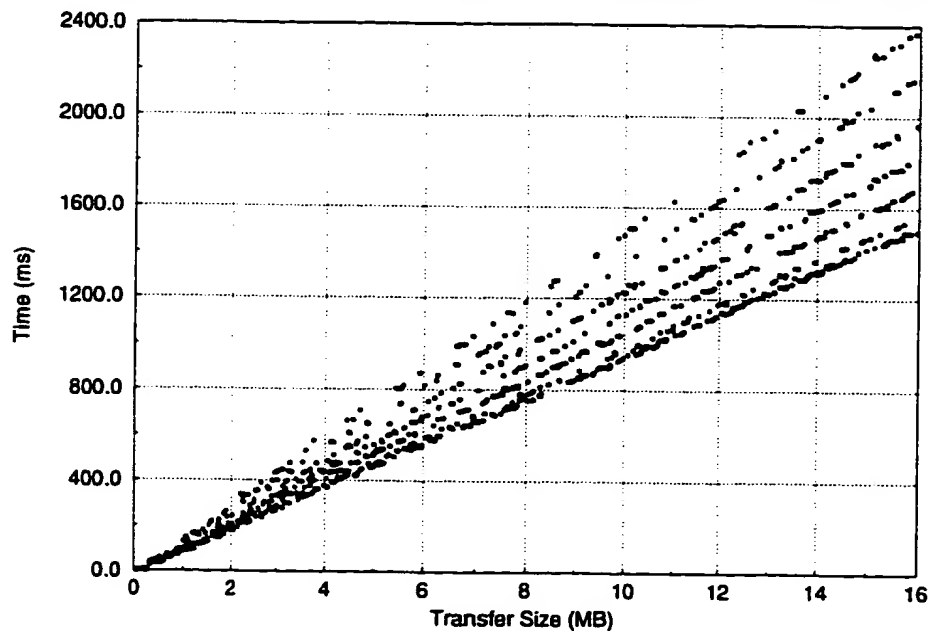


Figure 5.3: Seagate Barracuda 9 Random Access Write Performance. This figure plots write requests of random lengths to random locations on the disk. The plot shows a maximum transfer rate of 10.4 MB/s and a minimum access time of approximately 3 ms.

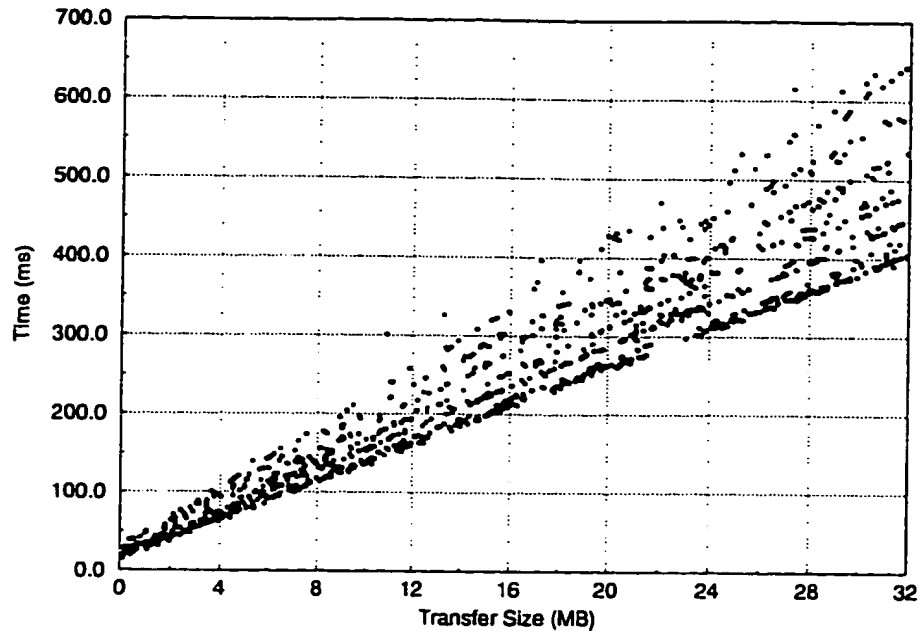


Figure 5.4: Ciprico 7010 Random Access Read Performance. This figure plots 4 KB aligned read requests of random lengths from random locations on the disk array. The plot shows a maximum transfer rate of 83 MB/s and a minimum access time of approximately 15 ms. Quantization of points into distinct lines results from accesses to different disk zones.

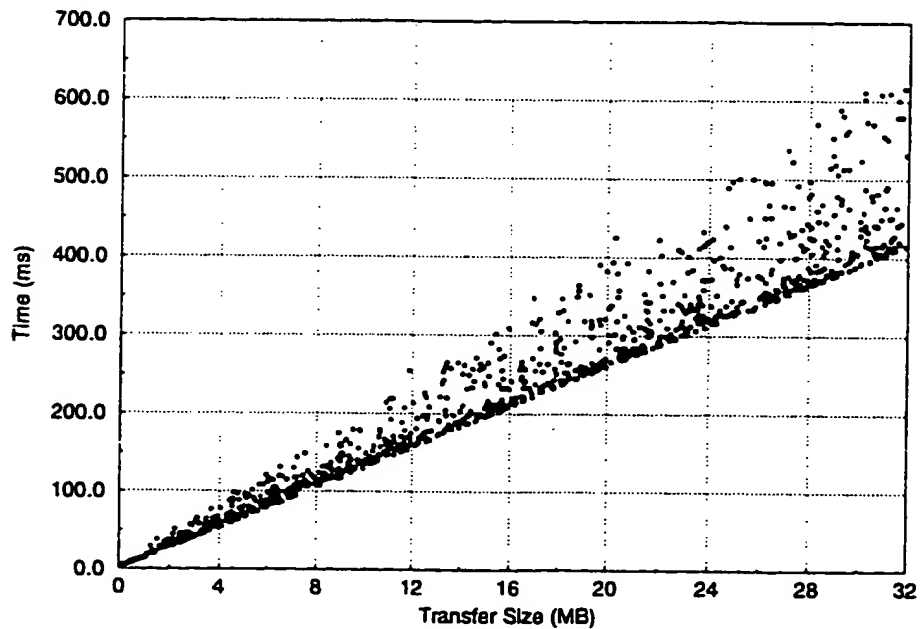


Figure 5.5: Ciprico 7010 Random Access Write Performance. This figure plots 4 KB aligned write requests of random lengths to random locations on the disk array. The plot shows a maximum transfer rate of 78 MB/s and a minimum access time of approximately 3 ms.

5.2.3 Sequential Access Performance of Raw Devices

This study conducts sequential access measurements by issuing several fixed-length device requests. Tests are only performed ten times, because variances between tests are small. This analysis uses arithmetic means of times to determine each transfer rate. All tests access outer disk zones to measure highest bandwidth performance.

Sequential access tests only time `read` or `write` systems calls. Each test opens the raw device, allocates memory, performs read or write operations, and closes the device. The read and write operations may include multiple `read` or `write` system calls.

Request sizes have strong influences on overall transfer rates. Given that each request has a fixed startup overhead and a nearly constant transfer rate, issuing fewer requests delivers the highest performance. Rates plateau where transfer times dominate startup times. Tests that transfer more data deliver higher transfer rates. The first request of these tests starts at block address 0 and may incur a track-to-track seek. With read-ahead caching, seeks from subsequent requests are negligible. Large transfers amortize the initial seek over a longer time period.

Sequential read performance is typically higher than write performance, due to read-ahead caching and zero-latency reads of the devices. Zero-latency reads begin to transfer data from the heads to disk buffers without waiting for the heads to rotate under the first disk sector. The devices do not support zero-latency writes.

Figures 5.6 and 5.7 plot request size versus transfer rate for 16 MB read and write raw sequential requests to Seagate Barracuda 9 disks. Figures 5.8 and 5.9 are similar plots for 256 MB transfers. All four figures plot transfer rates of a single disk and 2-wide, 4-wide, and 8-wide striped disks. The striping granularity is 64 KB.

The 256 MB transfers deliver slightly better performance than 16 MB transfers. The non-striped, 2-wide, and 4-wide tests provide flat transfer rate curves. Flat curves indicate small request startup overheads. Sequential read performance is usually higher than write performance.

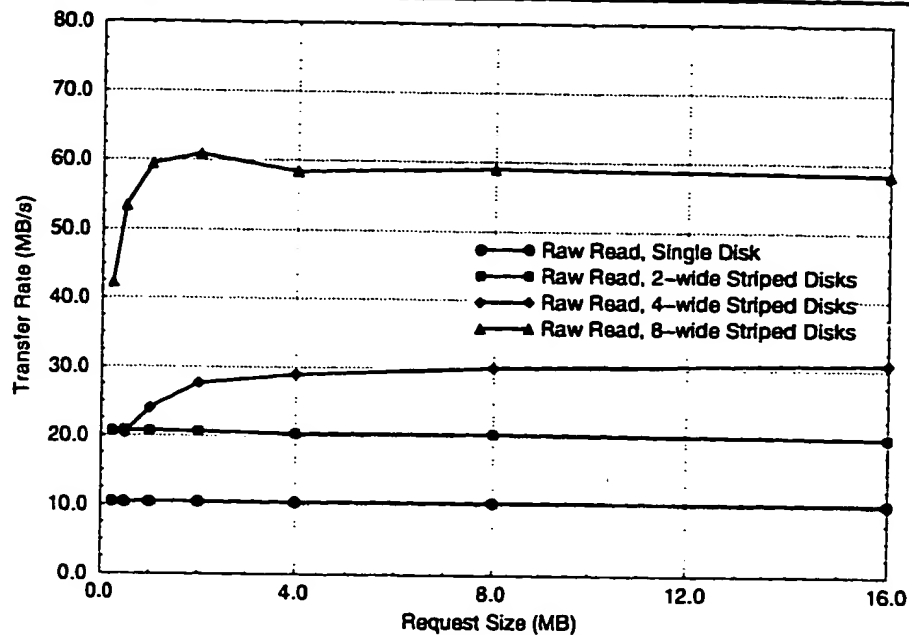


Figure 5.6: Seagate Barracuda 9 Sequential 16 MB Read Performance. The x-axis indicates the size of each request made to the disks while reading 16 MB. Plots express transfer rates for 1, 2, 4, and 8 disks.

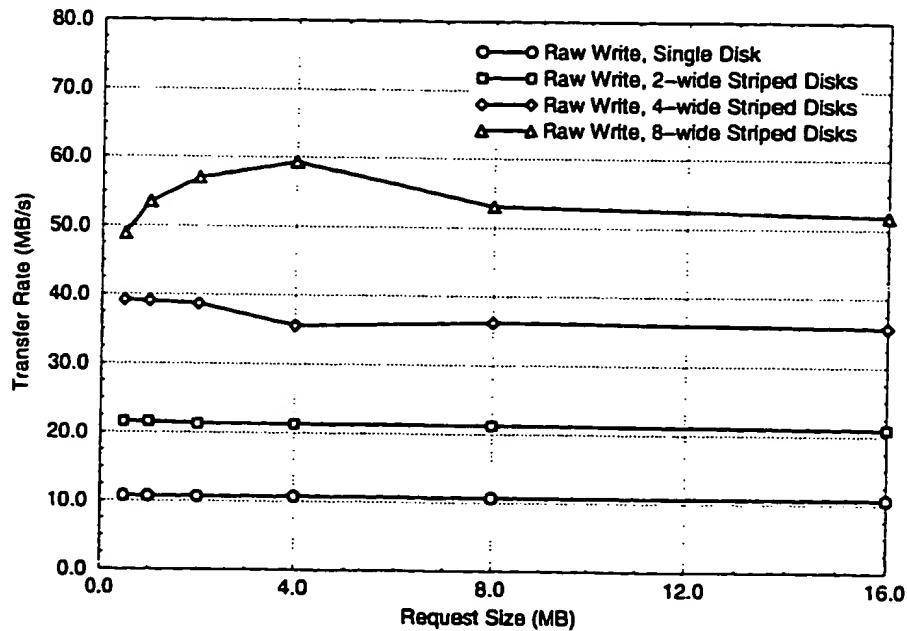


Figure 5.7: Seagate Barracuda 9 Sequential 16 MB Write Performance. The x-axis indicates the size of each request made to the disks while writing 16 MB. Plots express transfer rates for 1, 2, 4, and 8 disks.

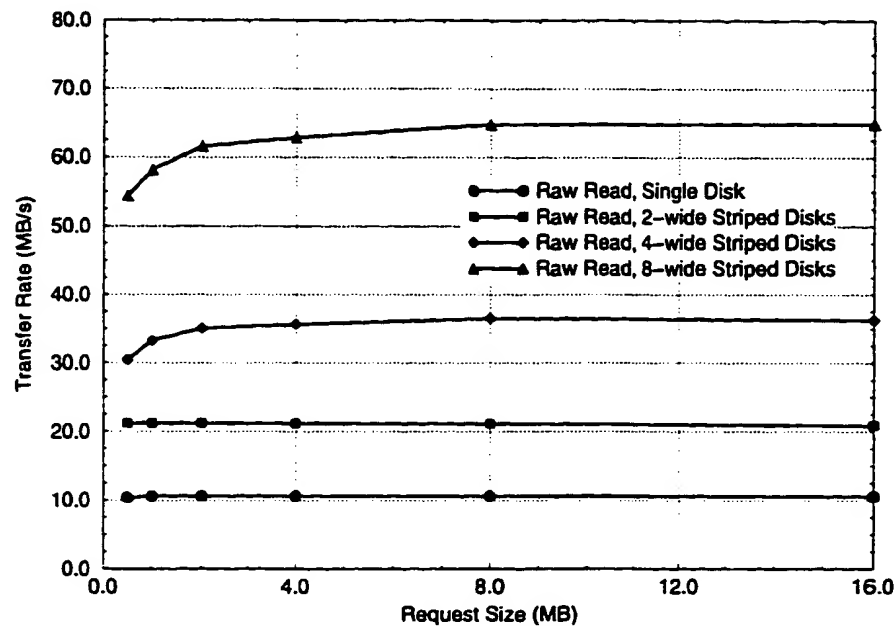


Figure 5.8: Seagate Barracuda 9 Sequential 256 MB Read Performance. The x-axis indicates the size of each request made to the disks while reading 256 MB. Plots express transfer rates for 1, 2, 4, and 8 disks.

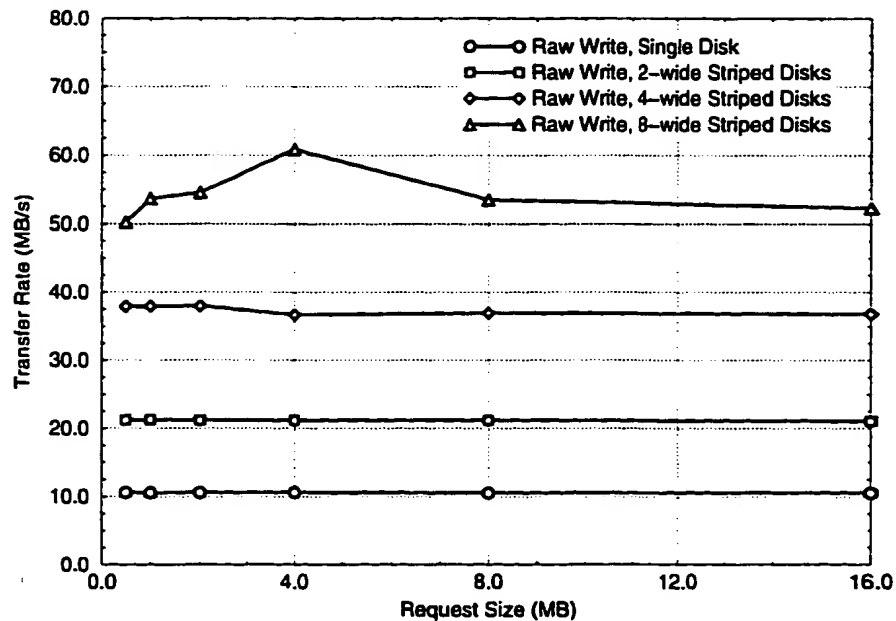


Figure 5.9: Seagate Barracuda 9 Sequential 256 MB Write Performance. The x-axis indicates the size of each request made to the disks while writing 256 MB. Plots express transfer rates for 1, 2, 4, and 8 disks.

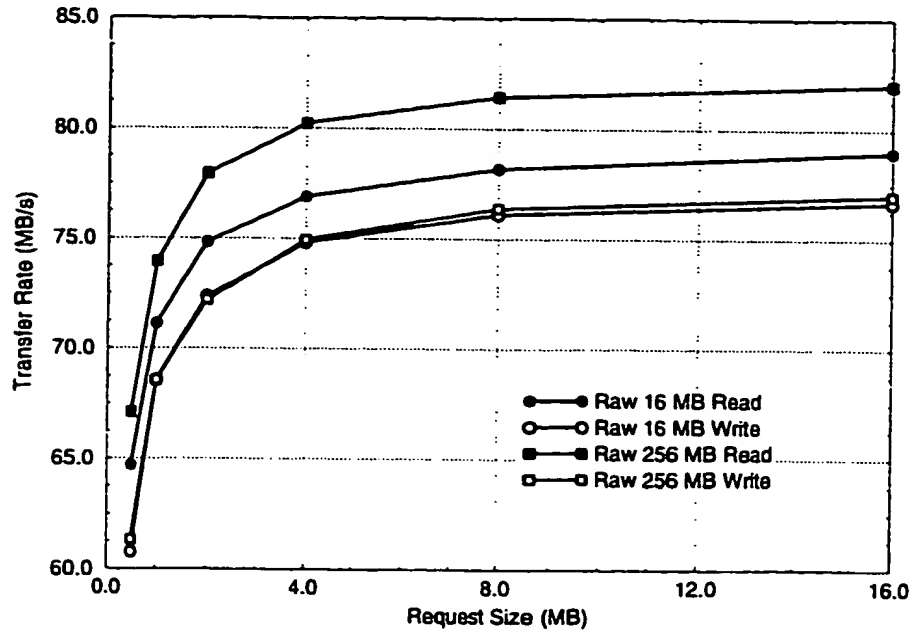


Figure 5.10: **Ciprico 7010 Sequential Read and Write Performance.** The x-axis indicates the size of each request made to the array while reading and writing 16 MB and 256 MB. The request size has a strong influence on the overall transfer rates, since each request has a fixed overhead. The curves tend to plateau after 8 MB request sizes.

Striped performance does not scale well after four devices. This poor scaling results, in part, from the software striping driver and the shared network port and loop. The SGI striping driver, *xlv*, introduces some overhead. Requests passing through this driver can lose as much as fifteen percent of the bandwidth as compared to requests bypassing the driver. Both serialization and contention cause performance degradation. The striping driver must serially send and receive device commands through a single Fibre Channel loop port. Devices competing for loop bandwidth cause contention.

Figure 5.10 plots request size versus transfer rate for raw sequential requests to a Ciprico disk array. The figure plots reads and writes performance for 16 MB and 256 MB transfers. In both cases, read performance is higher than write performance. The array performance curves tend to plateau at 8 MB request sizes. Striped disk performance curves are slightly flatter than disk array curves; however, disk array transfer rates are always higher for the entire range of these tests.

5.3 GFS Performance

This section investigates GFS performance. Although analysis presents transfer times and rates, the intent of the study is to compare these metrics with raw subsystem performance. The ratio of GFS performance to raw performance quantifies GFS client overheads. Single client performance is the baseline for comparisons of multiple client scaling. Tests configure multiple clients in a well balanced manner in order to ease analysis and aid repeatability. All clients simultaneously execute similar tests.

This study investigates file reads and writes. Analysis separates file writes into two cases: first time writes of new files and writes over existing files. These operations are referred to as create and write, respectively. In general, write performance is more similar to that of read performance than create performance, since writes and reads have nearly identical I/O request patterns. Creates require more metadata accesses than writes, in order to allocate blocks and update directories.

All GFS performance tests sequentially access files. In addition to timing `read` or `write` system calls, each test times `open` and `close` calls. Timings do not include memory allocation. The `open` system call performs directory lookups, vnode allocation, and inode allocation. The `close` system call releases the vnode and inode. The test program performs multiple fixed-length requests by issuing separate `read` and `write` system calls.

5.3.1 Single Client Performance

This section presents single GFS client performance. Tests investigate GFS performance on single Seagate disks, 8-wide striped disks, and single Ciprico disk arrays. These tests measure read, write, and create transfer times for several file sizes. Analysis reveals linear relationships between transfer time and file size. Furthermore, file performance varies with user request size.

Figure 5.11 plots file transfer times for a single Seagate disk and 8-wide striped disks. File sizes range from 0 bytes to 4 MB. All request sizes are less than 16 MB. The y-intercept represents the 0 byte file transfer time. The 0 byte file transfer time measures the time to open and close a file; no real data transfer occurs. The open/close time of GFS on Seagate disks is approximately 70 ms. Figure 5.12 plots transfer times for file sizes ranging from 0 bytes to 16 MB for a single disk and 0 bytes to 128 MB for 8-wide striped disks.

Figures 5.13 and 5.14 plot transfer times for a Ciprico disk array. Figure 5.13 plots file sizes from 0 bytes to 16 MB while Figure 5.14 plots file sizes up to 256 MB. Open/close times are approximately 65 ms.

These plots show a linear relationship between transfer time and file size. The y-intercept indicates the fixed overhead of opening and closing files. This overhead, consisting of a directory lookup and open file count manipulation, takes between 65 ms and 70 ms. For large files, the open/close overhead is insignificant; however, this overhead severely affects small file transfer times. File open/close times increase as the depths of the files within directories increase. All tests, unless otherwise specified, access files located in the file system root directory.

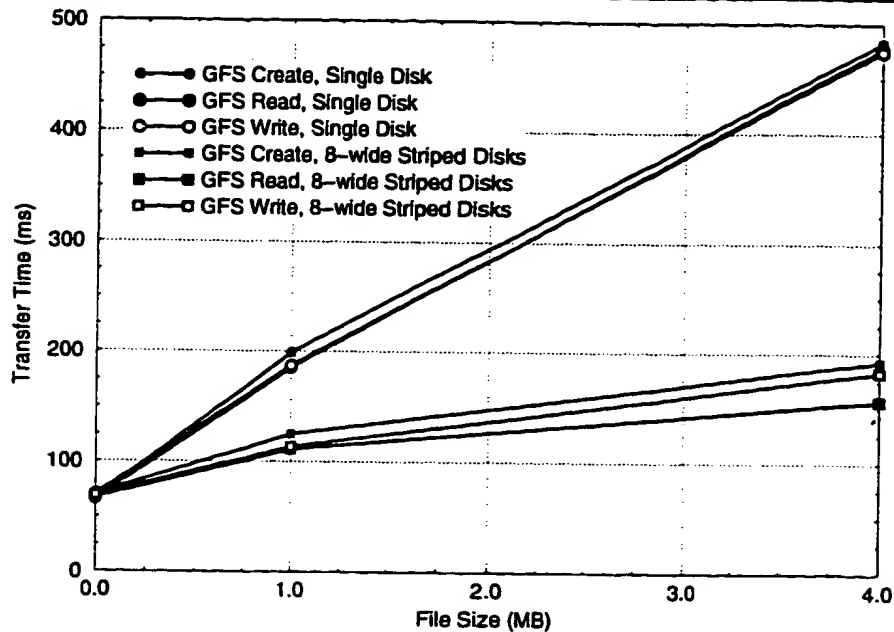


Figure 5.11: Transfer Times of GFS on Disks for File Sizes between 0 Bytes to 4 MB. This plot illustrates the time required to create, read, and write files of sizes up to 4 MB. Tests issue request sizes less than 4 MB. Results are given for a single disk and 8-wide striped disks. The y-intercept represents file open/close times. With respect to files smaller than 4 MB, the 70 ms open/close overhead is large.

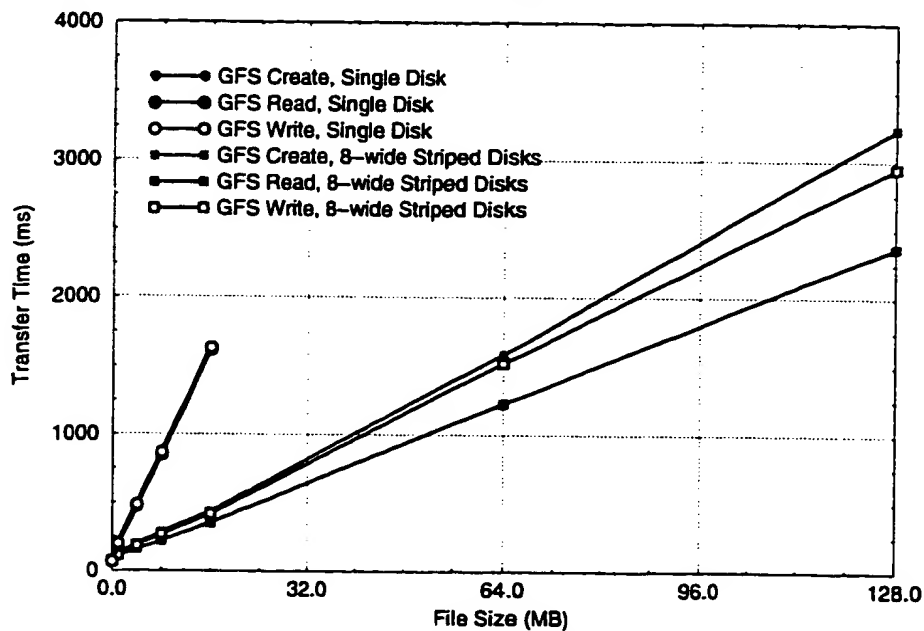


Figure 5.12: Transfer Times of GFS on Disks for File Sizes between 0 Bytes to 128 MB. This plot illustrates the time required to create, read, and write files of sizes up to 128 MB. Tests issue request sizes less than 16 MB. Results are given for a single disk and 8-wide striped disks. The 70 ms open/close overhead is negligible for files larger than 16 MB.

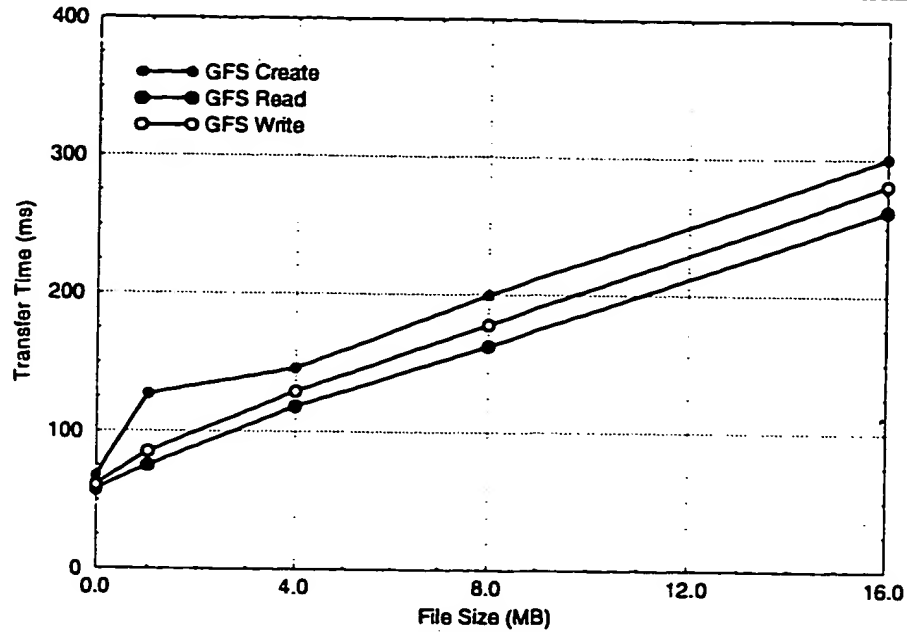


Figure 5.13: **Transfer Times of GFS on Arrays for File Sizes between 0 Bytes to 16 MB.** This plot illustrates the time required to create, read, and write files of sizes up to 16 MB. Tests issue request sizes less than 16 MB. Results are given for a single array. The y-intercept represents file open/close times. With respect to files smaller than 16 MB, the 65 ms open/close overhead is large.

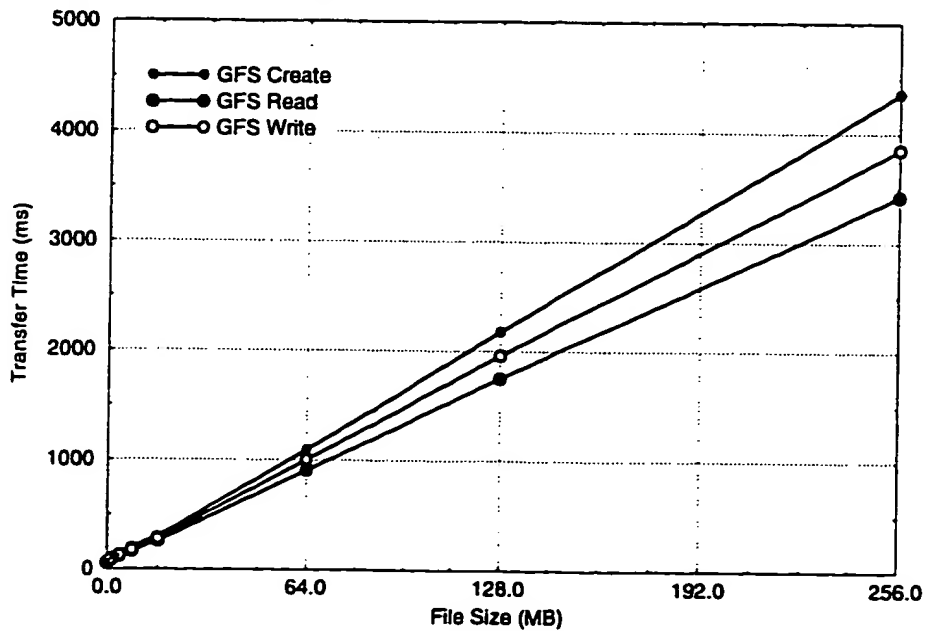


Figure 5.14: **Transfer Times of GFS on Arrays for File Sizes between 0 Bytes to 256 MB.** This plot illustrates the time required to create, read, and write files of sizes up to 256 MB. Tests issue request sizes less than 16 MB. Results are given for a single array. The 65 ms open/close overhead is negligible for files larger than 32 MB.

Figures 5.15 and 5.16 plot file transfer rates for various request sizes. These figures plot GFS performance on single disks and 8-wide striped disks during creates, reads, and writes of 4 MB and 16 MB files. Figures 5.17 and 5.18 illustrate file transfer rates of GFS on single Ciprico arrays for 16 MB and 256 MB files, respectively.

The x-axis of the plots indicate request sizes made to the file system. Larger request sizes deliver the highest transfer rates because fewer requests are issued. Like sequential accesses to raw devices, each file system request has fixed overheads. The time to acquire device locks and read metadata dominates file system request overheads. As a result, GFS transfer rates with small requests are slower than raw performance of similar request sizes.

In all cases, reads deliver the highest performance and creates yield the lowest. Slow create performance is primarily due to additional metadata accesses needed when growing files. File system write characteristics are slightly slower than read performance. Performance differences between reads and writes are a result of raw device characteristics and additional requests needed to update file access times during writes.

Like all file systems, GFS introduces overhead beyond that of raw device accesses. Overheads include directory lookups, metadata transfers, and external fragmentation. Local file systems which are capable of large scale caching may perform as well as raw transfers or possibly better, if user data is cached. In fact, the XFS local file system in SGI IRIX operating system delivers nearly the same performance as raw devices when a user application performs direct I/O. GFS is a distributed file system that must maintain consistency across multiple clients. This consistency comes at the cost of additional overheads.

This analysis estimates GFS overhead by comparing raw sequential performance with single client performance. Raw sequential device accesses deliver the best possible storage subsystem performance. Without caching, file systems cannot perform better than the raw sequential baseline. GFS makes a superset of requests with respect to raw sequential accesses. Additional requests include metadata reads and writes as well as device lock commands. With the exception of stuffed dinodes, the lack of real data caching ensures this superset relationship. Therefore, GFS comparisons to raw performance reveal GFS overhead.

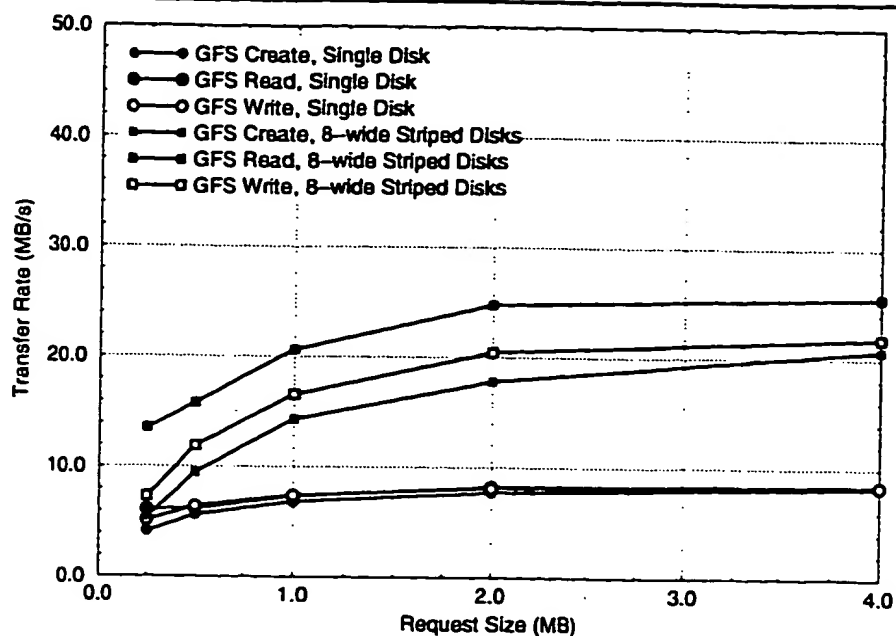


Figure 5.15: Transfer Rates of GFS on Disks for 4 MB Files. The x-axis indicates the size of each request made to the file system while creating, reading, and writing 4 MB files. Single disk and 8-wide striped disk measurements are given.

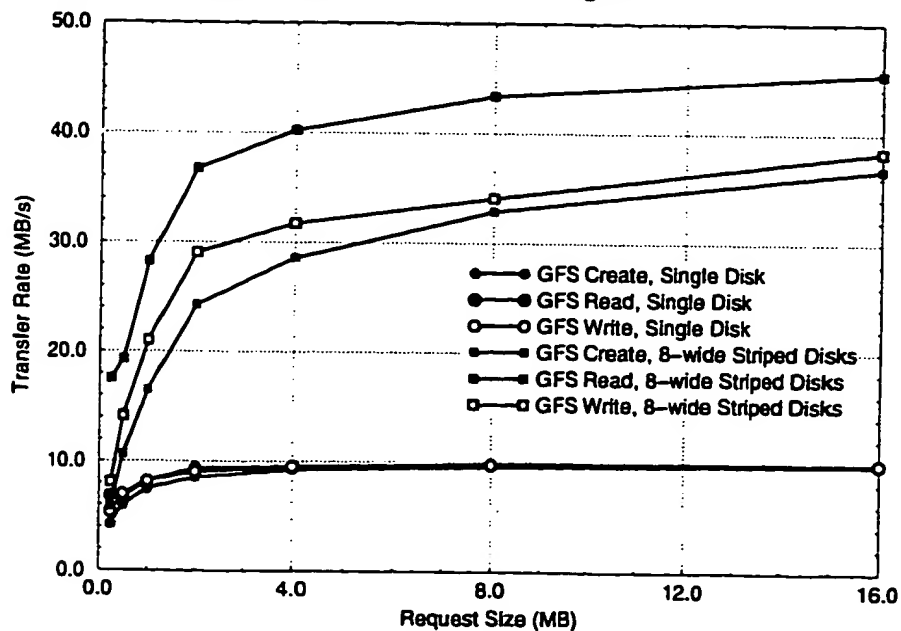


Figure 5.16: Transfer Rates of GFS on Disks for 16 MB Files. The x-axis indicates the size of each request made to the file system while creating, reading, and writing 16 MB files. Single disk and 8-wide striped disk measurements are given.

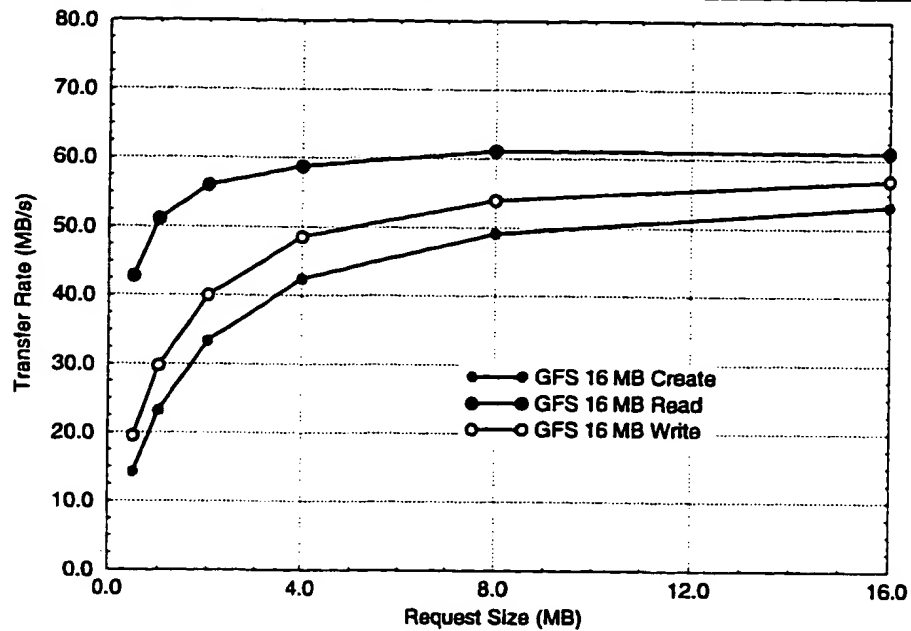


Figure 5.17: **Transfer Rates of GFS on Arrays for 16 MB Files.** The x-axis indicates the size of each request made to the file system while creating, reading, and writing 16 MB files. Single array measurements are given.

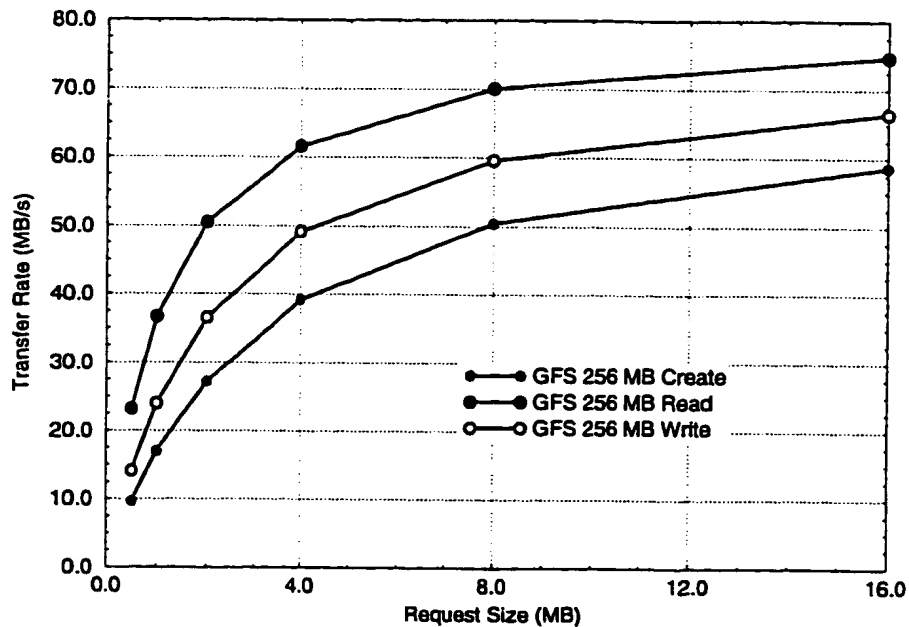


Figure 5.18: **Transfer Rates of GFS on Arrays for 256 MB Files.** The x-axis indicates the size of each request made to the file system while creating, reading, and writing 256 MB files. Single array measurements are given.

Tables 5.1, 5.2, and 5.3 summarize GFS overhead according to storage device type, file size, and request size. The ratio of GFS performance to raw performance represents GFS client efficiency. The difference between 100% and this ratio quantifies GFS overhead. This overhead does not include contention among clients.

Following the general trend, overheads are proportionately smaller for large files. This trend results from the large, fixed time of opening and closing files. The open/close time amortizes across the entire file transfer, so the open/close time is less significant for large files than small files. Furthermore, overhead grows as a function of request numbers, since GFS must maintain consistency as well as read and write metadata for each file request.

GFS overhead is between 5% and 15% for large file reads accessed with few requests. Large file sizes are relative to device speeds. For instance, a 16 MB file read with one request encounters an overhead of 5% on a single disk, 22% on 8-wide striped disks, and 23% on the disk array but has transfer rates of 9.9 MB/s, 45 MB/s, and 61 MB/s, respectively. GFS overhead for the disk array decreases to 9% as the file size increases to 256 MB. Fixed GFS overheads, such as opening and closing the file, proportionately affect high speed devices more than slow devices, because the data transfer times are shorter for high speed devices.

Sizes		Single Disk			8-Wide Disks			Disk Array		
File (MB)	Request (MB)	Raw (MB/s)	GFS (MB/s)	ratio (%)	Raw (MB/s)	GFS (MB/s)	ratio (%)	Raw (MB/s)	GFS (MB/s)	ratio (%)
4	2	10.5 ¹	8.22	78.3	60.9 ¹	24.8	40.7	74.8 ¹	33.5	44.8
4	4	10.4 ¹	8.38	80.6	58.5 ¹	25.5	43.6	76.9 ¹	33.8	44.0
16	8	10.5	9.81	93.4	59.1	43.4	73.4	78.2	61.1	78.1
16	16	10.4	9.90	95.2	58.5	45.4	77.6	79.0	61.0	77.2
128	8				64.8 ²	49.9	77.0	81.4 ²	68.5	84.1
128	16				64.9 ²	54.0	83.2	82.0 ²	73.1	89.1
256	8							81.4	70.1	86.1
256	16							82.0	74.8	91.2

¹ Estimated from 16 MB sequential performance tests.

² Estimated from 256 MB sequential performance tests.

Table 5.1: Raw Read vs GFS Read Performance This table summarizes GFS read overhead by comparing the ratio of GFS performance to raw, sequential read performance. The difference between 100% and the given ratio quantifies GFS overhead.

Sizes		Single Disk			8-Wide Disks			Disk Array		
File (MB)	Request (MB)	Raw (MB/s)	GFS (MB/s)	ratio (%)	Raw (MB/s)	GFS (MB/s)	ratio (%)	Raw (MB/s)	GFS (MB/s)	ratio (%)
4	2	10.7 ¹	7.73	72.2	57.1 ¹	17.8	31.2	72.4 ¹	23.2	32.0
4	4	10.7 ¹	8.28	77.4	59.3 ¹	20.8	35.1	74.8 ¹	27.4	36.6
16	8	10.7	9.62	89.9	53.1	32.8	61.2	76.1	49.1	64.5
16	16	10.6	9.80	92.5	51.9	36.7	70.1	76.7	53.2	69.9
128	8				53.5 ²	34.0	63.4	76.4 ²	51.1	66.9
128	16				52.3 ²	39.6	75.7	77.0 ²	59.0	76.7
256	8							76.4	50.4	66.0
256	16							77.0	58.7	76.2

¹ Estimated from 16 MB sequential performance tests.

² Estimated from 256 MB sequential performance tests.

Table 5.2: Raw Write vs GFS Create Performance This table summarizes GFS create overhead by comparing the ratio of GFS performance to raw, sequential write performance. The difference between 100% and the given ratio quantifies GFS overhead.

Sizes		Single Disk			8-Wide Disks			Disk Array		
File (MB)	Request (MB)	Raw (MB/s)	GFS (MB/s)	ratio (%)	Raw (MB/s)	GFS (MB/s)	ratio (%)	Raw (MB/s)	GFS (MB/s)	ratio (%)
4	2	10.7 ¹	8.12	75.9	57.1 ¹	20.4	35.7	72.4 ¹	27.5	38.0
4	4	10.7 ¹	8.42	78.7	59.3 ¹	21.8	36.8	74.8 ¹	31.0	41.4
16	8	10.7	9.76	91.2	53.1	34.1	64.2	76.1	54.0	71.0
16	16	10.6	9.91	93.5	51.9	38.3	73.8	76.7	57.0	74.3
128	8				53.5 ²	38.3	71.6	76.4 ²	58.7	82.0
128	16				52.3 ²	43.4	83.0	77.0 ²	65.4	84.9
256	8							76.4	59.6	78.0
256	16							77.0	66.5	86.4

¹ Estimated from 16 MB sequential performance tests.

² Estimated from 256 MB sequential performance tests.

Table 5.3: Raw Write vs GFS Write Performance This table summarizes GFS write overhead by comparing the ratio of GFS performance to raw, sequential write performance. The difference between 100% and the given ratio quantifies GFS overhead.

5.3.2 Multiple Client Aggregate Performance

This section studies multiple client aggregate performance. All clients simultaneously execute the same test benchmarks. Tests scale from a single client to four clients. Clients and disk arrays are added as pairs into the test environment. To exploit all parallelism in the distributed system, clients access files from different devices. However, contention for shared resources, like the root directory, exists. This analysis presents aggregate bandwidth and scaled speedup results.

Multiple client analysis investigates root directory contention using two test configurations. The first configuration places the root directory on a dedicated device. The complete test configuration includes three clients and four devices with one device exclusively devoted to the root directory. Clients encounter root directory contention caused by accesses from other clients. The second test configuration places the root directory on a device with file data. In addition to root directory contention, clients encounter contention caused by one client accessing files from the root directory device.

Figure 5.19 illustrates the dedicated root directory device configuration. All clients access the root directory located on array A. Arrays B, C, and D store files 0, 1, and 2, respectively; clients W, X, and Y access files 0, 1, and 2, respectively. Figure 5.20 illustrates the non-dedicated root directory device configuration. All clients access the root directory located on array A. Client W accesses file 0 from array A. Arrays B, C, and D store files 1, 2, and 3, respectively; clients X, Y, and Z access files 1, 2, and 3, respectively.

This analysis determines aggregate performance from the arithmetic mean of the slowest client time per test. Measurements reveal little deviation, even though each test executes only ten times. Because clients finish testing and remain idle until the last client completes, aggregate transfer rates are less than peak rates. This style of aggregate measurement is consistent with the Berkeley xFS performance study [56][71].

This analysis presents speedup values as the ratio of aggregate transfer rates of multiple clients to transfer rates of single clients. In this sense, the speedups are scaled, because problem sizes increase with the addition of each client and array. This ratio quantifies how GFS bandwidth scales as clients and storage devices are added. Scaled speedup is an appropriate measurement for this file system benchmarking, since these tests emphasize aggregate system bandwidth rather than response time of parallel applications.

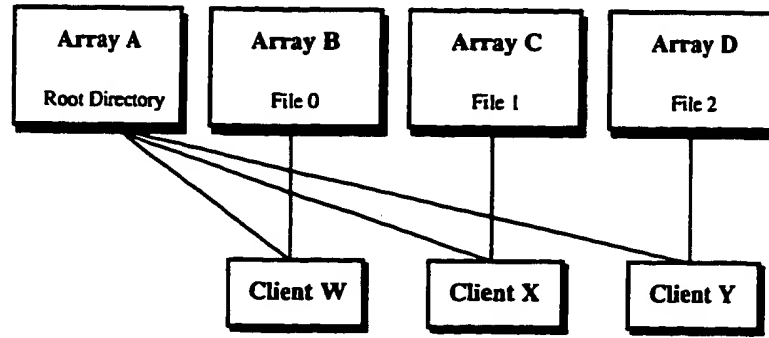


Figure 5.19: **Test Configuration with a Dedicated Root Directory Device.** This figure illustrates the file layout of the dedicated root directory device test configuration.

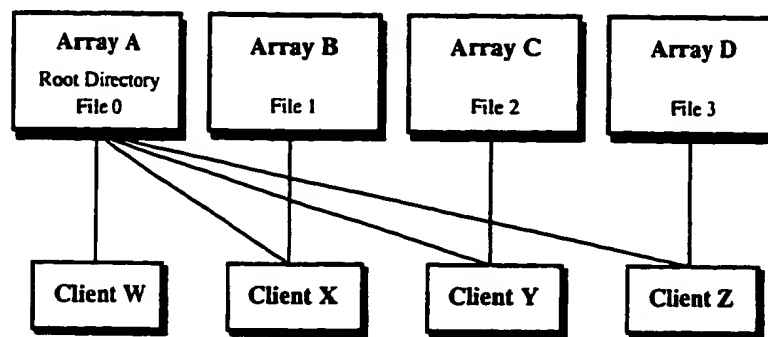


Figure 5.20: **Test Configuration with a Non-Dedicated Root Directory Device.** This figure illustrates the file layout of the non-dedicated root directory device test configuration.

Although experiments balance files among storage devices, resource groups are unbalanced. During benchmarking, the file system creation utility, *mkfs*, poorly mapped the resource groups to storage devices. Poor RG placement causes contention during file creations, since some devices contain two resource groups while other contained none. Unfortunately, the contention was not discovered until after benchmarking. Future work should balance RG placement during performance benchmarking.

Figure 5.21 plots aggregate transfer rate performance of 16 MB files with a dedicated root directory device. Figure 5.22 is the accompanying scaled speedup plot. Aggregate performance of three clients peaks at approximately 140 MB/s for both reads and writes. Scaled speedups for reads and writes are 2.3 and 2.5, respectively. This speedup is acceptable considering that single client performance is 78% and 65% of raw performance for reads and writes, respectively. Aggregate create bandwidth scales poorly to 2.1 for three clients, partially as a result of unbalanced resource group placement.

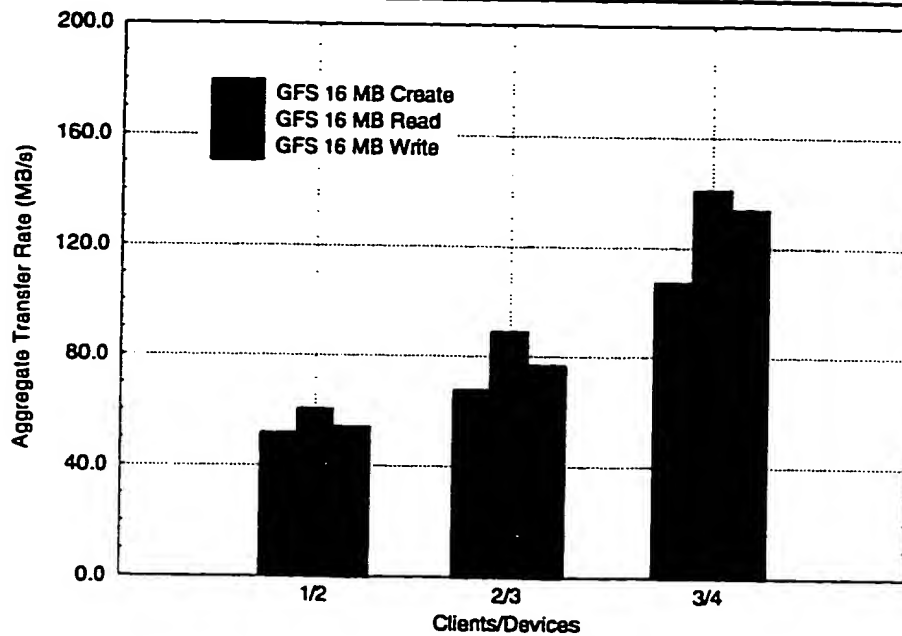


Figure 5.21: Aggregate Transfer Rates for 16 MB Files with a Root Directory Device. This plot illustrates aggregate transfer rates of three clients simultaneously creating, reading, and writing 16 MB files with 8 MB requests. The root directory exclusively occupies one array; three other arrays contain file data.

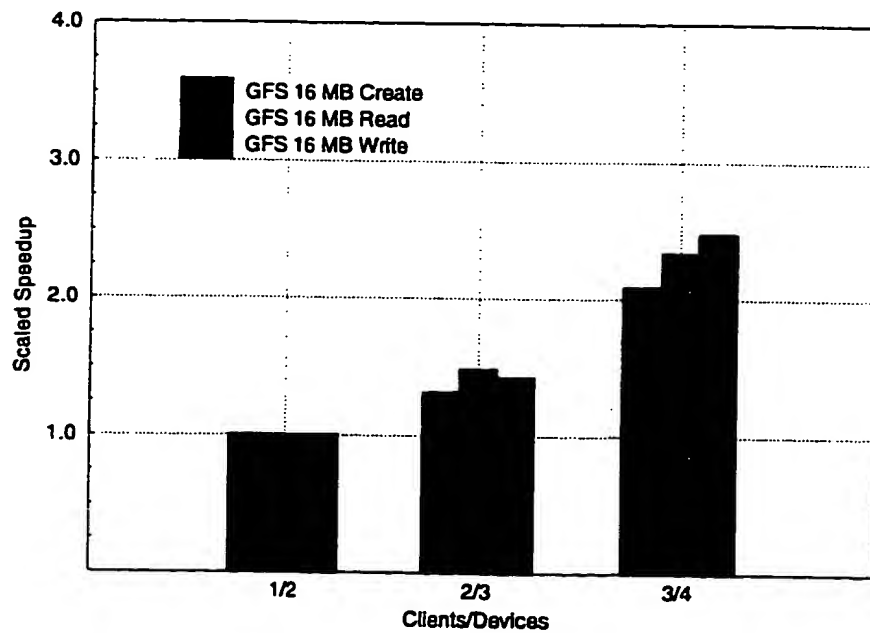


Figure 5.22: Scaled Speedups for 16 MB Files with a Root Directory Device. This plot illustrates scaled speedups of three clients simultaneously creating, reading, and writing 16 MB files with 8 MB requests.

Figures 5.23 and 5.24 plot aggregate transfer rates and scaled speedups for 256 MB file performance with a dedicated root directory device. For large file transfers, contention of the root directory is less critical. The aggregate performance of three clients is 200 MB/s and 159 MB/s for reads and writes, respectively. The speedup scales almost linearly for both tests. Create performance does not scale beyond 2.

Figures 5.25 and 5.26 plot aggregate transfer rates and scaled speedups for 16 MB file performance in the non-dedicated root directory device test configuration. The aggregate performance of four clients is 127 MB/s for reads and 121 MB/s for writes. These results show speedups of 1.7, 2.1, and 2.2 for creates, reads, and writes, respectively. The non-dedicated root directory device tests do not perform as well as tests with a dedicated root directory device. Poor scaling is due to contention between client W requests to file 0 and other client requests to the root directory.

Figures 5.27 and 5.28 show aggregate transfer rates and scaled speedups for 256 MB file performance in the non-dedicated root directory device test configuration. These tests scale poorly with respect to the dedicated device tests. Plots reveal that aggregate performance decreases with the addition of a fourth client. Once again, this characteristic results from root directory device contention. Performance peaks at three clients with an aggregate transfer rate of 180 MB/s for reads and 152 MB/s for writes. The scaled speedup for three clients is 2.6 and 2.5 for reads and writes, respectively. Create performance does not scale beyond 1.7 for two or more clients.

Clearly, contention for shared resources causes low speedups. The performance tests of this section are demanding, since all clients simultaneously begin file accesses and the analysis calculates aggregate performance based on slowest client times. Clients racing to access the root directory cause contention. Tests without a dedicated root directory also incur contention between file and directory accesses. In the non-dedicated device case, client directory requests may wait for the completion of 8 MB file requests from client W. The dedicated root directory device eliminates this contention. Results show large file performance scales almost linearly.

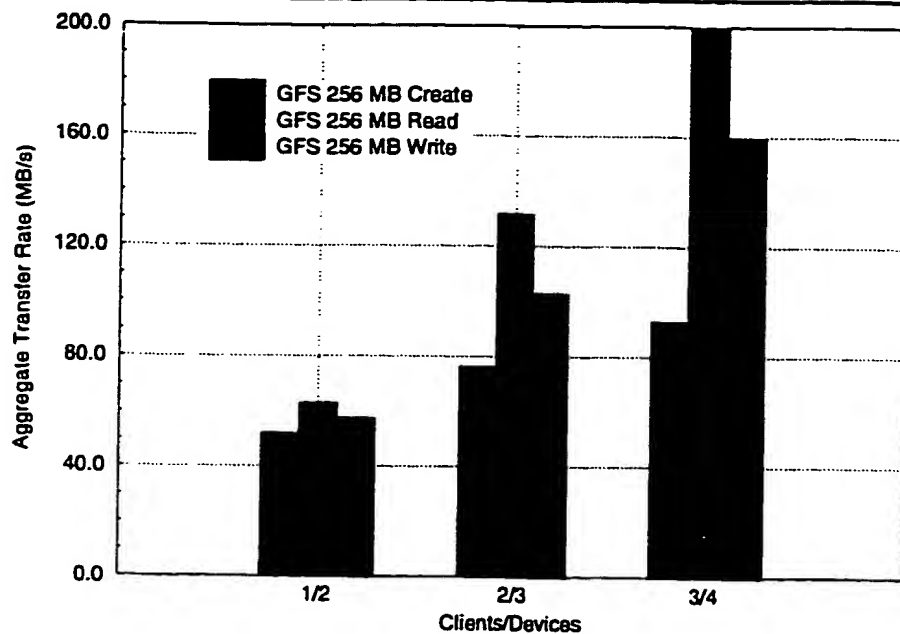


Figure 5.23: Aggregate Transfer Rates for 256 MB Files with a Root Directory Device. This plot illustrates aggregate transfer rates of three clients simultaneously creating, reading, and writing 256 MB files with 8 MB requests. The root directory exclusively occupies one array; three other arrays contain file data.

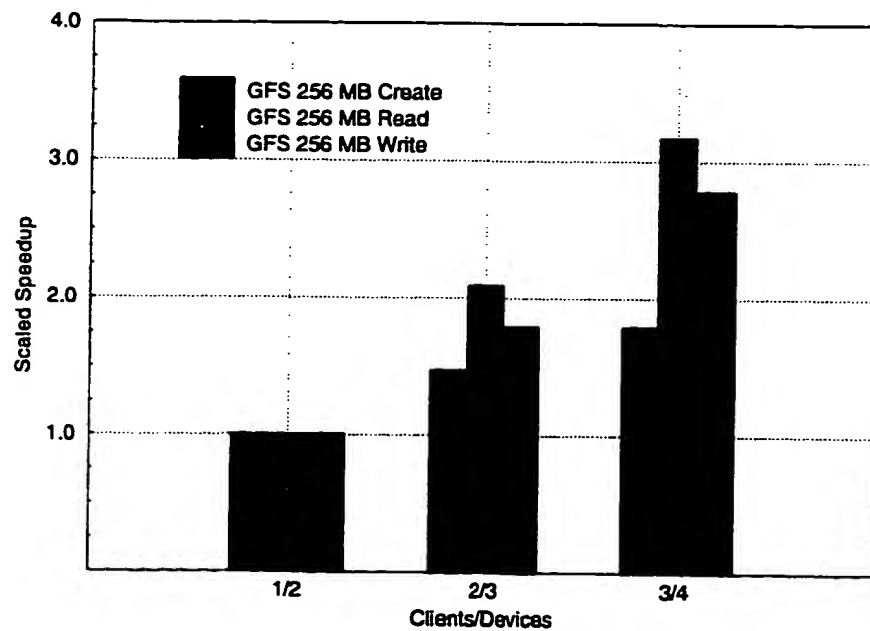


Figure 5.24: Scaled Speedups for 256 MB Files with a Root Directory Device. This plot illustrates scaled speedups of three clients simultaneously creating, reading, and writing 256 MB files with 8 MB requests.

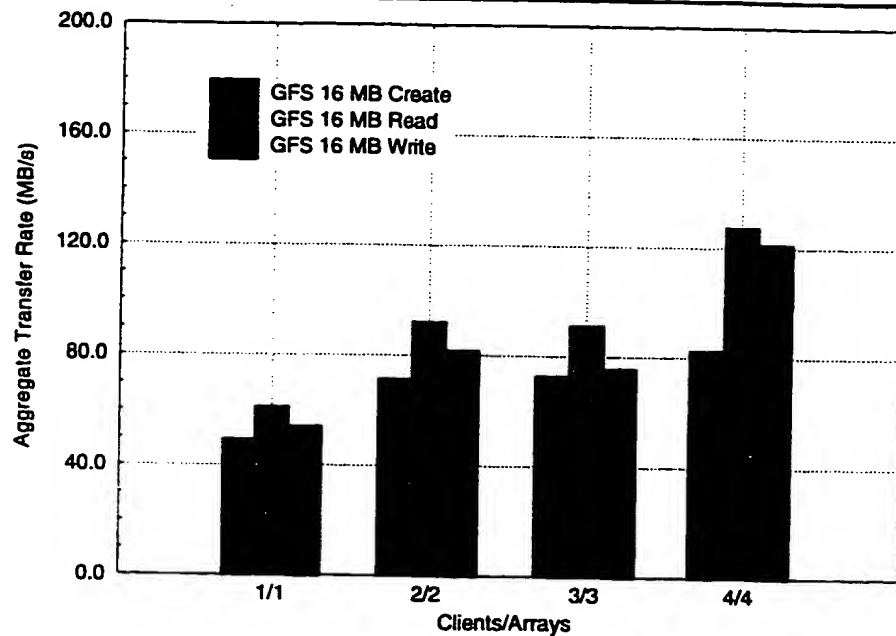


Figure 5.25: **Aggregate Transfer Rates for 16 MB Files without a Root Directory Device.** This plot illustrates aggregate transfer rates of four clients simultaneously creating, reading, and writing 16 MB files with 8 MB requests. The root directory shares an array with file data.

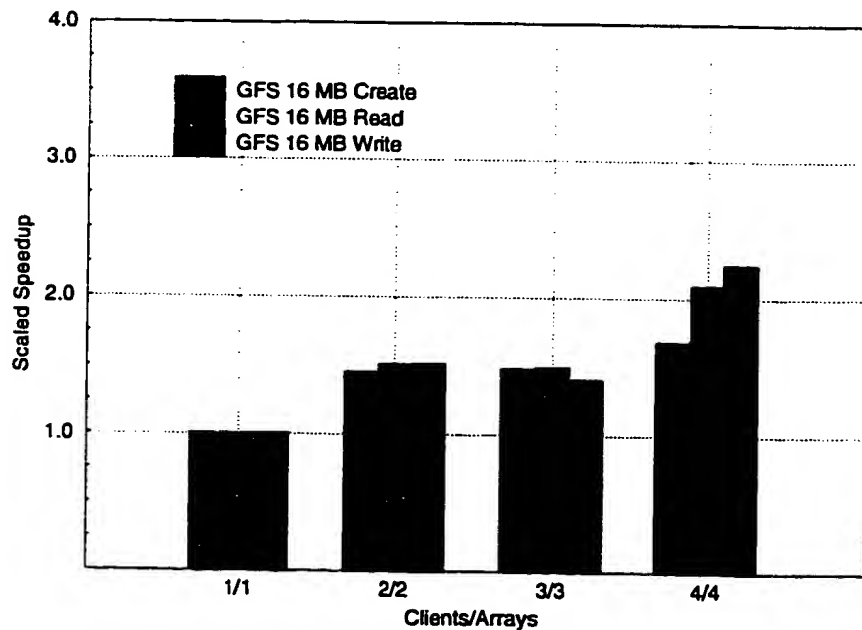


Figure 5.26: **Scaled Speedups for 16 MB Files without a Root Directory Device.** This plot illustrates scaled speedups of four clients simultaneously creating, reading, and writing 16 MB files with 8 MB requests.

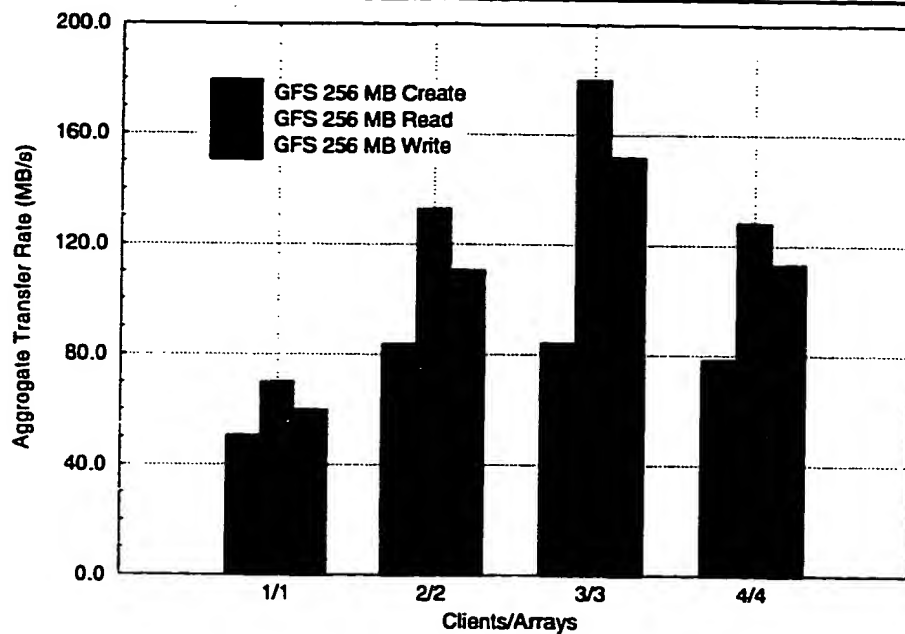


Figure 5.27: Aggregate Transfer Rates for 256 MB Files without a Root Directory Device. This plot illustrates aggregate transfer rates of four clients simultaneously creating, reading, and writing 256 MB files with 8 MB requests. The root directory shares an array with file data.

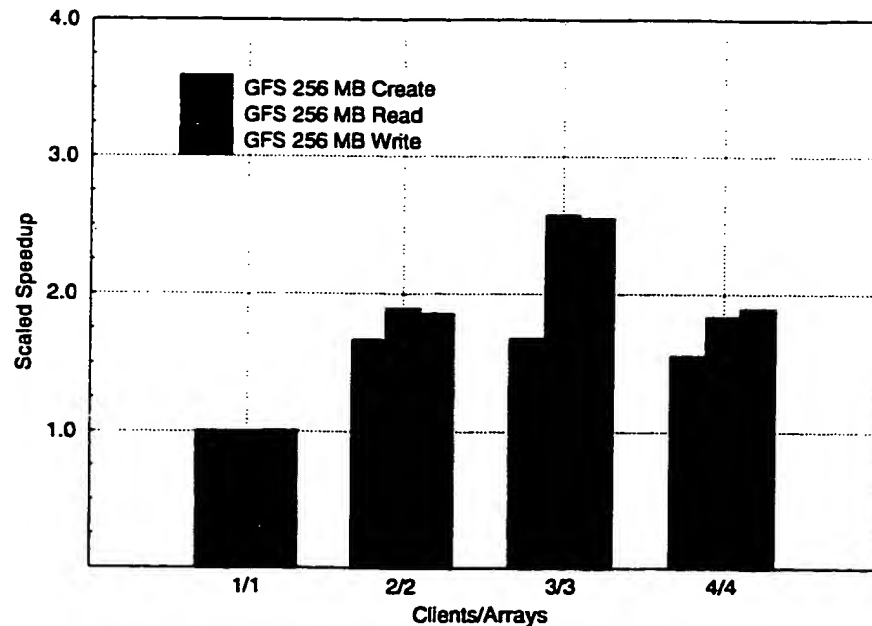


Figure 5.28: Scaled Speedups for 256 MB Files without a Root Directory Device. This plot illustrates scaled speedups of four clients simultaneously creating, reading, and writing 256 MB files with 8 MB requests.

5.3.3 Multiple Client Throughput Performance

The previous section analyzes performance of multiple clients simultaneously executing the same tests. Every client performs the same request pattern and must compete for shared resources at nearly the same time. While this benchmark appropriately models some applications, a more randomized workload models general file system activity. Unlike previous tests, clients in these random tests do not wait for other clients to finish file transfers. Clients continuously transfer files throughout each test. As a result, these tests measure file system throughput.

Clients access private directories located in the file system root directory. Prior to benchmarking, files for read and write operations are created in these private directories. During benchmarking, clients write newly created files to the private directories. To free space, clients remove files immediately after creation.

File operation mixtures are a distribution of 70% reads, 15% writes, and 15% creates. The tests uniformly distribute file sizes between 0, 1, 2, 4, 8, 16, 32, 64, and 128 MB. With a maximum request size of 8 MB, clients access files using a uniform distribution of 1, 2, 4, 8, and 16 requests. This file size distribution favors large request sizes. For instance, clients only access 128 MB files with sixteen 8 MB requests. Clients access 64 MB files with either sixteen 4 MB requests or eight 8 MB requests. The purpose of these distributions is to provide a range of file activity rather than model a real-world environment. Two thirds of file accesses are to files 16 MB or smaller.

Figure 5.29 plots the speedup of clients accessing GFS with a random workload. These tests execute on configurations with and without dedicated root directory devices. Each test runs for one hour. The ratio of the amount of data transferred by multiple clients to that of a single client determines speedup.

The speedup plot shows sub-linear speedup, indicating resource contention. For the dedicated root directory tests, accesses to the root directory as well as poorly balanced resource groups cause contention. File accesses to the root directory device cause additional contention in the non-dedicated root directory tests. Nevertheless, speedup of three machines with a dedicated root directory device is 2.4 and speedup of four machines with a non-dedicated root directory device is 2.6. With two-thirds of the operations accessing files 16 MB or smaller, each test encounters a great deal of contention.

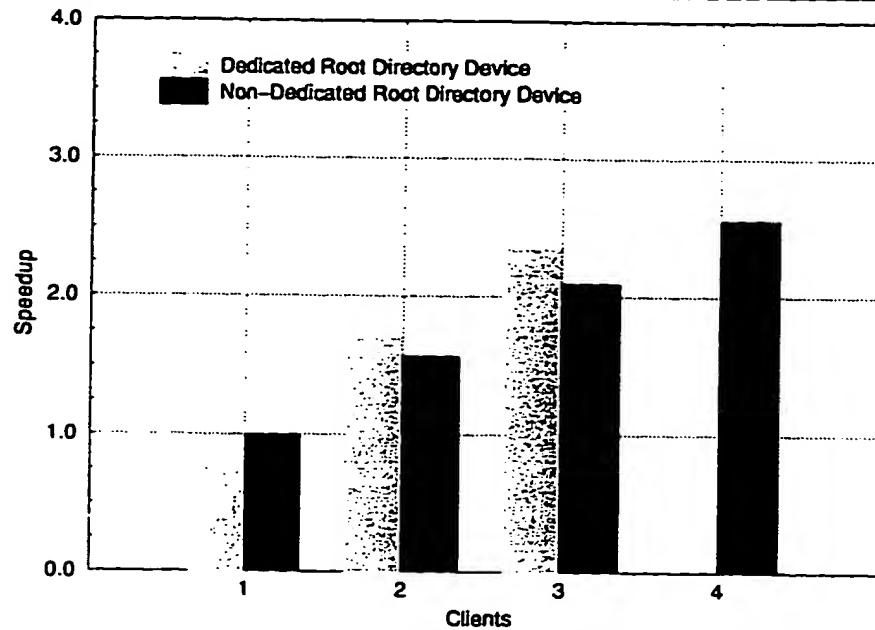


Figure 5.29: **Throughput Scaling with Random Workload.** This plot represents throughput speedups of clients accessing GFS with random workloads. The plot presents results from tests configurations with and without dedicated root directory devices.

5.3.4 Directory Performance

Directory depth directly influences GFS file lookup times. All tests so far, except throughput tests, access files on file system root directories. This section measures the time needed to access files at different directory depths. Each directory depth traversal requires a minimum of locking a directory dinode, verifying access, reading the dinode, and unlocking the dinode.

Figure 5.30 shows access times of existing files at various directory depths. The test environment consists of a single SGI Challenge and a single Seagate disk or Ciprico disk array. The x-axis of the plot indicates directory tree depth. The root directory has a depth of 0. The y-intercept represents the time to open and close a file. Open and close operations lock file dinodes, read dinodes, increment or decrement file open counts, write dinodes to storage, and unlock dinodes. The Seagate disk cache acts only as a read ahead buffer, so disk writes, while opening and closing files, take longer than the Ciprico disk array.

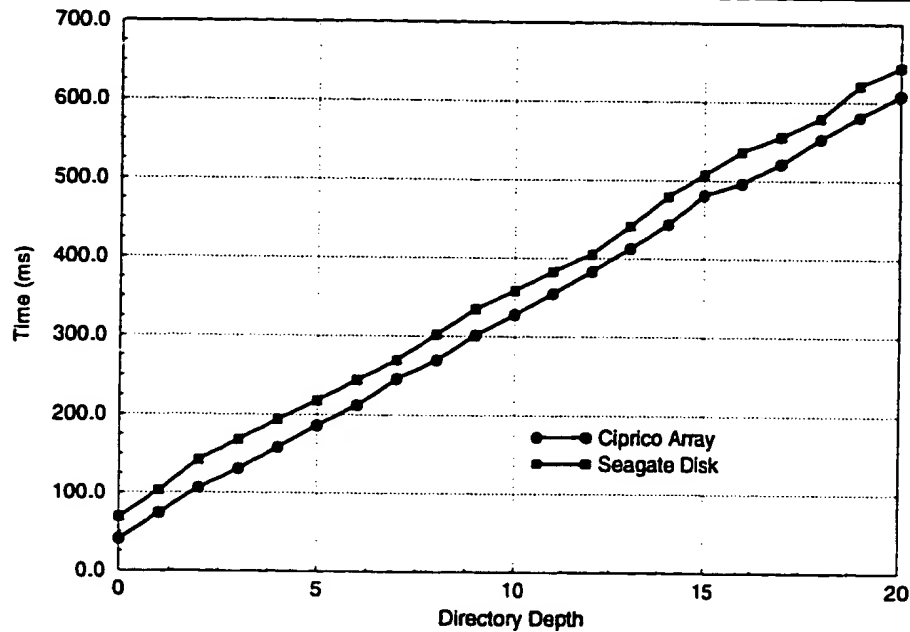


Figure 5.30: Directory Performance. This plot represents the time required to open and close existing files located at various directory depths. The plot illustrates times for single Seagate disks and Ciprico arrays.

The plots illustrates a linear relationship between time and directory depth. Given this linear relationship, the slope of the plotted line reveals the time of each directory traversal. Each directory traversal takes approximately 28 ms. This large directory lookup overhead is devastating to small file performance though affects large file performance less. The current GFS implementation only caches open files. During these tests, the file system only caches the root directory. Directory caching is fundamental to directory traversal performance. Future implementations of GFS should incorporate directory caching.

5.4 Summary

This chapter provides basic performance analysis of the Global File System. Performance tests focus on sequential file accesses with file sizes greater than 1 MB. The analysis compares performance of single clients with raw device performance. Single client performance is the baseline for multiple client scaling characterizations. A summary of GFS performance analysis follows:

- A linear relationship between file transfer time and file size exists. Substantial overhead occurs when opening and closing files. This overhead is insignificant for files larger than 8 MB for single disks and 32 MB for striped disks or disk arrays.
- GFS single client overheads range between 5% and 70% below raw sequential device performance. Overheads are small for large files and large request sizes. Overheads are larger on high-bandwidth devices than low-bandwidth devices for equal file and request sizes.
- Each file request incurs larger overheads than requests to raw devices. Accessing entire files with few requests delivers higher performance than accessing files with many requests.
- The root directory is a source of multiple client contention. File transfer performance with a dedicated root directory device is significantly better than performance of the non-dedicated root directory device configuration. Lock requests and I/O transfers cause root directory device contention. Large file performance scales linearly up to three clients and four devices, if the file system dedicates one device for the root directory. Random throughput tests scale well for both dedicated and non-dedicated root directory device configurations.
- Directory lookup times are directly proportional to directory tree depths.

File system evaluation is inherently difficult, because so many factors influence performance. These factors include the numbers of clients and devices, data layout, file fragmentation, and workloads. Furthermore, a detailed quantitative comparison between GFS and existing systems is not yet feasible. Fibre Channel is so new that not many existing systems have been ported or tuned for Fibre Channel. Comparisons based on existing systems like NFS on Ethernet are not realistic, since the workloads tested in this section tend to overwhelm such configurations. Tests of NFS on 100 MB/s HiPPI deliver a disappointing 5 MB/s with a dedicated NFS server and client. Even though HiPPI performance is comparable to FC, NFS is not tuned for such a network or workload. Consequently, the most informative comparisons are between GFS and the raw subsystem.

Chapter 6

Conclusions

It's more fun to arrive at a conclusion than to justify it.

– Malcolm S. Forbes

Ethernet and SCSI are two interface standards that have increased performance and connectivity overtime while maintaining backwards compatibility. Fibre Channel is an emerging standard that out performs both Ethernet and SCSI in bandwidth and connectivity. Because users are often reluctant pay a premium price to upgrade existing systems without receiving an order of magnitude increase in performance, Fibre Channel may have difficulty competing with existing standards. However, the combined channel and network traits of Fibre Channel is a competitive advantage that may propel this emerging standard into commodity markets.

Network attached storage support is a significant strength of Fibre Channel. FC storage devices directly attach to networks and are shared by all connected computers. Due to direct access, the data path streamlines. Clients directly access shared data in a more distributed, less complex system, thereby obviating the need to hide data behind overworked servers. Fibre Channel emerges as the only competent interface that facilitates high performance, high availability distributed file systems based upon shared network storage.

6.1 A Solution to Poor Distributed File System Performance

Existing distributed file systems fail to deliver adequate performance in large storage capacity environments. This dissertation addresses these performance difficulties through modeling and measurement. Chapter 3 presents a first-order model that reveals throughput

performance benefits from network attachment of block-addressable storage devices. The model reveals that network attached storage architectures potentially deliver storage device performance without loss. Fibre Channel is an existing interface that supports network attached storage.

The Global File System breaks the client-server tradition of distributed file systems. GFS exploits the channel interface of Fibre Channel to provide the shortest data path between clients and shared storage devices. GFS clients use private file managers to service local requests. GFS differs from existing network attached storage file systems in that GFS is designed from scratch as a distributed file system rather than a modified local file system. Only ground-up development reasonably addresses caching, metadata serialization, and file system layout techniques.

GFS prototype benchmarks confirm that file systems based upon shared network storage deliver performance comparable to raw storage subsystems. Scalability measurements suggest that private file manager designs and parallelized file system structures exhibit good parallelism. Design refinements could improve performance to levels that satisfy currently impractical applications.

6.2 Future Directions

This dissertation addresses basic design issues for distributed file systems that possess private file managers and use network attached storage. Initial performance measurements of the prototype are promising and warrant future research. Five critical areas need additional attention: (1) cross-platform implementation, (2) additional performance benchmarking, (3) small file performance, (4) scalability, and (5) recovery mechanisms for all failure types.

6.2.1 Cross-Platform Implementations

The GFS architecture supports cross-platform compatibility. Although the prototype is only SGI IRIX compatible, future implementations should include other platforms. Two cross-platform compatibility issues that need addressing are data structure encoding and device label formats.

For compatibility, all platforms must encode data structures using consistent byte ordering and data type sizing. Currently, GFS defines data structures in terms of byte lengths

and big-endian byte ordering. All big-endian platforms recognize data in the same manner. However, implementations on little-endian platforms must translate to the big-endian scheme.

The second obstacle to cross-platform compatibility is device label formats. Device labels store configuration information required by device drivers. Currently, labeling schemes are platform specific. Device labeling schemes need standardization.

6.2.2 Additional Performance Benchmarking

Additional benchmarking is essential to further characterize GFS performance. More testing could quantify areas that need improvement, such as small file transfers. Future benchmarking should also expand scalability tests by increasing numbers of clients and devices. Furthermore, tests should include multiple processes on each machine.

Currently, reasonable comparisons to competing technologies are not practical. To make meaningful comparisons, hardware configurations must be similar. As network protocols like IP become available on Fibre Channel, new possibilities for GFS comparisons will arise.

Benchmarking provides a controlled environment for performance analysis; however, synthetic workloads are no substitute for real-world applications. Applications could reveal problems and bottlenecks that benchmarks cannot. Applications may require greater stability than GFS currently delivers. Further development should stress stability issues.

To date, the most significant application run on GFS is a technology demonstration at the 1997 National Association of Broadcasters (NAB) convention. For this demonstration, GFS served short video movies to SGI graphics workstations. Ciprico disk arrays were connected through a Brocade FC switch to GFS clients. The movies played at thirty frames per second and required sixty megabytes per second file system performance. The demonstration ran for more than eight hours each day during the four day convention. The NAB demonstration shows performance capabilities of the GFS prototype called from a demanding, yet well controlled, application. Future applications should include random workloads that perform a mixture of reads, writes, and directory accesses.

6.2.3 Improving Small File Performance

Benchmark tests reveal inadequate performance on files smaller than one megabyte. Since most directories are only a few kilobytes in length, poor small file performance remains a serious issue. Traversing several directories, in order to locate files, adversely affects file performance regardless of file size. Poor small file performance of the prototype, at least in part, is a result of minimal client and device caching.

The current implementation only caches data from open files. Since directories are rarely open longer than the duration of a lookup operation, directories seldom remain cached. Thus, most directory searches cannot take advantage of client caching and must read the directory from storage. The current implementation should be enhanced to cache data using a replacement policy such as LRU.

Reducing storage device access times would further improve small file performance. Data frequently used by multiple clients is not always cached in device caches. Based on data type, GFS currently determines which requests might benefit from device caching. The file system tags these requests for caching and passes each request to the NSP device driver. However, lower level drivers do not propagate cache control information to devices. Lower level drivers should be augmented to support directive caching.

Current device cache sizes range from 256 KB to 1 MB. Quantifying cache hit rates based upon cache sizes and file system access patterns is an area of potential research. This research could also investigate various cache replacement policies.

Solid state devices may further improve small file performance. Solid state devices service requests with access times similar to device caches. Small files and metadata from large files are good candidates for placement on solid state devices. Presently, Fibre Channel solid state devices are not available.

6.2.4 Improving Scalability

The aggregate performance of distributed file systems should either sustain or increase as machines and devices are added. Without good scalability, poor performance overshadows benefits gained by distributed file sharing. Two factors that affect scalability are resource sharing and data layout. Directories are often shared resources, so techniques designed to improve small file performance tend to improve scaling. Resource groups attempt to parallelize accesses to shared resources. Benefits and limitations of resource groups need further investigation.

The performance study, presented in Chapter 5, balances data by placing files on user specified resource groups. Real-world applications do not have this luxury, so the technique must be automated. Possible schemes include randomly allocating files to different resource groups, allocating users to specific resource groups, and dynamically moving files from heavily-loaded to lightly-loaded resource groups.

Device lock improvements could also increase scalability. Currently, device lock commands are not reordered in device command queues. Two command reordering schemes would reduce lock and unlock command latencies; the first scheme would reorder unlock commands while the second would reorder lock commands. Both schemes assume synchronous transfers and lock commands. The first scheme would place unlock commands before all read, write, and lock requests on the command queue. By performing such a reordering, pending lock requests that may have otherwise failed would succeed. In addition, this reordering could reduce unlock request latencies. The second scheme would reorder lock commands before all read and write requests, but behind unlock commands. This reordering would slightly affect read and write request performance. However, lock performance would improve since locks would not queue behind slower I/O requests. Because command queue scheduling is partially controlled by device drivers, the NSP device driver and lower level Fibre Channel drivers should be modified to include both reordering schemes.

6.2.5 Recovery Mechanisms

File system recovery is essential in distributed systems. Component failures must leave file systems in recoverable states. Furthermore, recovery must be timely and not affect functional components. While network and device redundancies reduce the risks of file system failure, computer failures can leave metadata in inconsistent states.

Chapter 4 presents a recovery mechanism for device locks. This mechanism guarantees that locks do not permanently remain in locked states. However, lock recovery does not ensure that data associated with the locks is consistent. Repair utilities, similar to the UNIX *fsck* utility, may salvage file systems after failures that corrupt metadata. Repair tool run times, however, are often proportional to file system sizes, so repairing large file systems may be impractical.

Further research needs to solve file system inconsistencies caused by client and device failures. Three inconsistency types that occur when clients fail are: (1) open file dinodes

and pointer trees may be corrupted, (2) open file counts are never decremented, and (3) files locked with mandatory and advisory locks remain locked.

A possible scheme that solves all three inconsistencies requires active clients to clean-up after failed clients. Clients would be assigned unique identifiers at mount time. Upon opening files or setting file locks, clients would store this mount identifier within a list on each dinode. Clients would remove the identifiers from the dinode lists after unlocking or closing files. Failed clients could not remove identifiers from dinodes.

Active clients would maintain lists of failed client mount identifiers. When opening files, clients would search dinode identifier lists for failed client entries. Upon discovering failed client entries, active clients would remove these entries from dinode lists and decrement file open counts. Then, active clients would search through the file metadata for inconsistencies left by failed clients. If possible, clients would repair these inconsistencies; otherwise, the file would be removed.

Manager processes running in client user spaces would compile lists of mount identifiers corresponding to failed clients. These manager processes would periodically pass updated lists to kernel-level file systems. Managers would build these lists from information stored in the file system superblock. Superblocks would contain lists of mounted machine names and associated identifiers as well as lists of active identifiers. Managers would identify active and failed machines by using a heart-beat protocol and the two superblock lists. This scheme would distribute recovery functionality to all participating clients without influencing normal operation. Only files affected by failed clients would need to be verified and repaired. File open count and file lock inconsistencies would be resolved without much difficulty.

Other possible recovery techniques include log-structured metadata and commit protocols. Log-structuring metadata is a technique that would write metadata changes to a log on disk. This technique is based on checkpoints, so consistency would be re-established by stepping backward through the log. Log-structuring may complicate normal file system activity, but this technique could simplify recovery.

A commit protocol could be designed such that devices do not save a chain of write requests until the last request is received. The current implementation of GFS assembles metadata modifications and then successively writes the data to storage. Using a commit strategy, GFS would write the chain of modifications to storage followed by the commit command. No single modification would be written to stable storage until the chain is committed. In

this way, changes to metadata would be atomic. Given enough device memory, a commit strategy may be less complex than log structured approaches.

6.3 Summary

This dissertation presents a distributed file system design which shares network attached storage devices among independent client computers. With private file managers and devices directly serving clients, a great deal of complexity is removed from the file system design. Network attached storage architecture provides the shortest path between storage and user memory. Performance from this direct data access is comparable to that of raw storage devices. GFS delivers performance beyond the capabilities of current distributed file systems. Such performance enables applications that were once restricted to local file systems to run in distributed environments.

Bibliography

- [1] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System," *Proceedings of the Summer USENIX Conference*, pp. 119–130, 1985.
- [2] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer*, pp. 9–20, May 1990.
- [3] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, "Measurements of a Distributed File System," *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 198–212, October 1991.
- [4] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the 10th ACM Symposium on Operating System Principles*, pp. 15–24, December 1985.
- [5] E. Riedel and G. Gibson, "Understanding Customer Dissatisfaction With Underutilized Distributed File Servers," *Proceedings of the Fifth NASA/Goddard Conference on Mass Storage Systems and Technologies*, pp. 371–387, September 1996.
- [6] P. Woodward, "Interactive Scientific Visualization of Fluid Flow," *IEEE Computer*, pp. 13–25, October 1993.
- [7] R. Braham, "The Digital Backlot," *IEEE Spectrum*, pp. 51–63, July 1995.
- [8] A. S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 2 ed., 1989.
- [9] G. A. Gibson *et al.*, "A Case for Network-Attached Secure Disks," tech. rep., Carnegie Mellon University, June 1996. CMU-CS-96-142.
- [10] ANSI X3T9.2 Committee, *SCSI-2 Specification*, January 31, 1994. X3.131-1994.
- [11] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 2 ed., 1990.

- [12] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *IEEE Computer*, pp. 17–28, March 1994.
- [13] R. Karedla, J. S. Love, and B. Wheery, "Caching Strategies to Improve Disk System Performance," *IEEE Computer*, pp. 38–46, March 1994.
- [14] E. Grochowski and R. F. Hoyt, "Future Trends in Hard Disk Drives," *IEEE Transactions on Magnetics*, vol. 32, no. 3, pp. 1850–1854, May 1996.
- [15] S. Miastkowski, "World's fastest disk drive," *BYTE*, p. 48, January 1997.
- [16] Seagate Technology Inc., *Cheetah Family 3.5-inch Form-Factor*, 1997.
<http://www.seagate.com/disc/cheetah/cheetah.shtml>.
- [17] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, "RAID: High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, no. 2, pp. 145–185, June 1994.
- [18] G. Ganger, B. Worthington, R. Hou, and Y. Patt, "Disk Arrays: High-Performance, High-Reliability Storage Subsystems," *IEEE Computer*, pp. 30–36, 1994.
- [19] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *International Conference on Management of Data (SIGMOD)*, pp. 109–116, June 1988.
- [20] D. Deming, *The SCSI Tutor*. Saratoga, CA: ENDL Publications, 1994.
- [21] J. Stai, *The SCSI Bench Reference*. Saratoga, CA: ENDL Publications, 1992.
- [22] P. Steenkiste, "A Systematic Approach to Host Interface Design for High-Speed Networks," *IEEE Computer*, pp. 47–57, March 1994.
- [23] J. D. Day and H. Zimmermann, "The OSI Reference Model," *Proceedings of the IEEE*, vol. 71, pp. 1334–1340, December 1983.
- [24] IEEE, New York, 802.3: *Carrier Sense Multiple Access with Collision Detection*, 1985.
- [25] IEEE, New York, 802.4: *Token-Passing Bus Access Method*, 1985.
- [26] IEEE, New York, 802.5: *Token Ring Access Method*, 1985.

- [27] DARPA Internet Program, Arlington, VA, *Internet Protocol*, September 1981. RFC 791.
- [28] DARPA Internet Program, Arlington, VA, *Transmission Control Protocol*, September 1981. RFC 793.
- [29] J. Postel, *User Datagram Protocol*. USC/Information Sciences Institute, August 1980. RFC 768.
- [30] R. H. Katz, "High-Performance Network and Channel Based Storage," *Proceedings of the IEEE*, vol. 80, no. 8, pp. 1238–1261, August 1992.
- [31] M. Sachs, A. Leff, and D. Sevigny, "LAN and I/O Convergence: A Survey of the Issues," *IEEE Computer*, pp. 24–32, December 1994.
- [32] R. Van Meter, "A Brief Survey of Current Work on Network Attached Peripherals," *ACM Operating Systems Review*, pp. 63–70, January 1996.
- [33] A. F. Benner, *Fibre Channel: Gigabit Communication and I/O for Computer Networks*. New York, NY: McGraw Hill, 1996.
- [34] M. Sachs and A. Varma, *IBM Research Report: Fibre Channel*. IBM Research Division, November 1995.
- [35] C. Jurgens, "Fibre Channel: A connection to the future," *IEEE Computer*, pp. 88–90, August 1995.
- [36] M. Bach, *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986.
- [37] T. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of the Summer 1986 USENIX Conference*, pp. 238–247, June 1986.
- [38] A. Silberschatz, J. Peterson, and P. Galvin, *Operating Systems Concepts*. Reading, MA: Addison-Wesley Publishing Company, Inc., fourth ed., 1994.
- [39] G. Pajari, *Writing UNIX Device Drivers*. Reading, MA: Addison-Wesley Publishing Company, Inc., 1992.
- [40] W. R. Stevens, *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1990.

- [41] Sun Microsystems, Inc, Mountain View, CA, *XDR: External Data Representation Standard*, June 1987. RFC 1014.
- [42] B. Goodheart and J. Cox, *The Magic Garden Explained*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1994.
- [43] U. Vahalia, *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1996.
- [44] A. S. Tanenbaum, *Distributed Operating Systems*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1995.
- [45] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems*. New York, NY: McGraw-Hill, Inc., 1994.
- [46] M. J. Tucker, "NFS Accelerators," *SunExpert Magazine*, pp. 59–64, August 1996.
- [47] P. Valduriez, "Parallel Database Systems: the case for shared-something," *Proceedings of the Ninth International Conference on Data Engineering*, pp. 460–465, 1993.
- [48] G. F. Pfister, *In Search of Clusters*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1995.
- [49] M. Satyanarayanan, "A Survey of Distributed File Systems," tech. rep., Carnegie Mellon University, 1989.
- [50] E. Levy and A. Silberschatz, "Distributed File Systems: Concepts and Examples," *ACM Computer Surveys*, vol. 22, no. 4, pp. 321–374, DECEMBER 1990.
- [51] B. Parwlowshi *et al.*, "NFS Version 3: Design and Implementation," *Proceedings of the Summer USENIX Conference*, 1994.
- [52] J. Ousterhout, A. Cherenon, F. Douglass, M. Nelson, and B. Welch, "The Sprite Network Operating System," *IEEE Computer*, pp. 23–36, February 1988.
- [53] V. Srinivasan and J. Mogul, "Spritely NFS: Experiments with Cache-Consistency Protocols," *Proceedings of the 12th ACM Symposium on Operating System Principles*, pp. 45–57, 1989.
- [54] M. Satyanarayanan, "Coda: A Highly Available File System for a Distributed Workstation Environment," *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, September 1989.

- [55] S. Chutani *et al.*, "The Episode File System," *Proceedings of the Winter 1992 USENIX Conference*, pp. 43–60, 1992.
- [56] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, "Serverless Network File System," *ACM Operating Systems Review*, vol. 29, no. 5, December 1995.
- [57] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, pp. 26–52, February 1992.
- [58] M. Rosenblum, *The Design and Implementation of a Log-Structured File System*. PhD thesis, University of California at Berkeley, 1992.
- [59] J. Hartman and J. Ousterhout, "Zebra: A Striped Network File System," *USENIX Workshop on File Systems*, May 1992.
- [60] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson, "A Quantitative Analysis of Cache Policies for Scalable Network File Systems," *Proceedings of the 1994 SIGMETRICS Conference*, May 1994.
- [61] M. Dahlin, *Serverless Network File System*. PhD thesis, University of California at Berkeley, 1995.
- [62] N. Kronenberg, H. Levy, and W. Strecker, "VAXclusters: A Closely-Coupled Distributed System," *ACM Transactions on Computer Systems*, vol. 4, no. 2, pp. 130–146, May 1986.
- [63] Digital Technical Journal, *VAXcluster Systems*, September 1987. Special Issue – Number 5.
- [64] D. Wiltzius and K. Minuzzo, "Network-attached peripherals (NAP) for HPSS/SIOF," tech. rep., Lawrence Livermore National Laboratory, 1995. Available at http://www.llnl.gov/liv.comp/siof/siof_nap.html.
- [65] R. W. Watson and R. A. Coyne, "The Parallel I/O Architecture of the High-Performance Storage System (HPSS)," *14th IEEE Symposium on Mass Storage Systems*, pp. 27–44, September 1995.
- [66] IEEE Storage System Standards Working Group (SSSWG) (Project 1244), "Reference Model for Open Storage Systems Interconnection, Mass Storage Reference Model Version 5," September 1994.

- [67] K. Matthews, "Implementing a Shared File System on a HIPPI Disk Array," *Fourteenth IEEE Symposium on Mass Storage Systems*, pp. 77–88, 1995.
- [68] M. Devarakonda, A. Mohindra, J. Simoneaux, and W. Tetzlaff, "Evaluation of Design Alternatives for a Cluster File System," *1995 USENIX Technical Conference*, January 1995.
- [69] T. Berners-Lee *et al.*, *Hypertext Transfer Protocol – HTTP/1.0*. Network Working Group, May 1996. RFC 1945.
- [70] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *1996 USENIX Technical Conference*, January 1996.
- [71] M. Dahlin, "Personal Communication." E-mail message, February 1997.